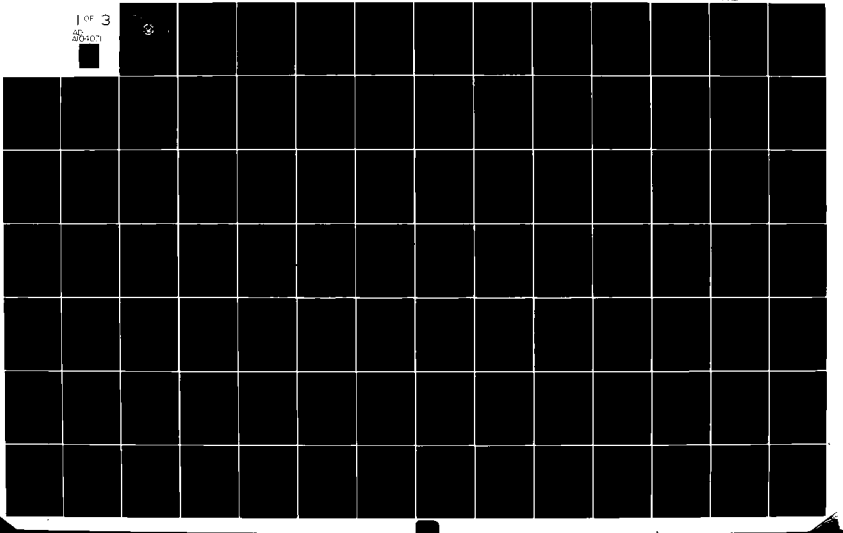
AD-A104 071 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
DETAILED DESIGN AND IMPLEMENTATION OF THE KERNEL OF A REAL-TIME--ETC(U)
MAR 81 D K RAMANTZIKOS

UNCLASSIFIED

NL

1 OF 3
AD A104071



LEVEL II

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD A104071

DTIC
ELECTE
SEP 1 1 1981
D
H

THESIS

Detailed Design and Implementation of the Kernel of a Real-Time Distributed Multiprocessor Operating System

by

Demosthenis Konstantinos Rapantzikos

March 1981

Thesis Advisor: T. F. Tao
Thesis Advisor: U. R. Kodres

Approved for public release; distribution unlimited

DTIC FILE COPY

81 9 11 013

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A104 071	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Detailed Design and Implementation of the Kernel of a Real-Time Distributed Multiprocessor Operating System,		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis March, 1981
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Demosthenis Konstantinos Rapantzikos		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE March, 1981
		13. NUMBER OF PAGES 282
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Operating systems, distributed computer networks, microprocessors, real-time processing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This thesis presents the detailed design and implementation of the kernel of a real-time, distributed operating system for a microcomputer based multiprocessor system.

Process oriented structure, segmented address spaces and a synchronization mechanism based on eventcounts and sequencers comprise the central concepts around which this operating system is built.

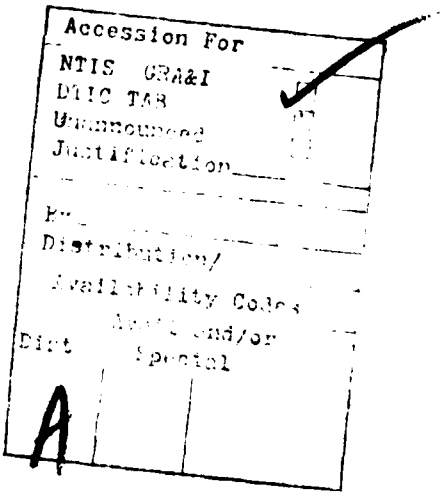
The operating system is hierarchically structured, layered in three loop free levels of abstraction and fundamentally configuration independent. This design

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 (cont'd) permits the logical distribution of the kernel functions in the address space of each process and the physical distribution of system code and data among the microcomputers. This physical distribution in turn, in a multimicroprocessor configuration will help to minimize system bus contention.

The system particularly supports applications where processing is partitioned into a set of multiple interacting asynchronous processes. One such application is that of smart sensor image processing for which this system has been specifically developed. The implementation was developed for the INTEL 86/12A single-board computer using the 8086 processor chip.



Approved for public release; distribution unlimited

Detailed Design and Implementation of the Kernel of a Real-Time Distributed Multiprocessor Operating System

by

Demosthenis Konstantinos Rapantzikos
Lieutenant, Hellenic Navy
Greek Naval Academy, 1969

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

and

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March, 1981

Author: _____

Approved by:

_____
Thesis Advisor

_____
Thesis Advisor

_____
Second Reader

_____
Second Reader

_____
Chairman, Department of Computer Science

_____
Chairman, Department of Electrical Engineering

_____
Dean of Information and Policy Sciences

_____
Dean of Science and Engineering

ABSTRACT

This thesis presents the detailed design and implementation of the kernel of a real-time, distributed operating system for a microcomputer based multiprocessor system.

Process oriented structure, segmented address spaces and a synchronization mechanism based on eventcounts and sequencers comprise the central concepts around which this operating system is built.

The operating system is hierarchically structured, layered in three loop free levels of abstraction and fundamentally configuration independent. This design permits the logical distribution of the kernel functions in the address space of each process and the physical distribution of system code and data among the microcomputers. This physical distribution in turn, in a multimicroprocessor configuration will help to minimize system bus contention.

The system particularly supports applications where processing is partitioned into a set of multiple interacting asynchronous processes. One such application is that of smart sensor image processing for which this system has been specifically developed. The implementation was developed for the INTEL 86/12A single-board computer using the 8086 processor chip.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I am indebted to and especially grateful to Professors T. Tao, U. Kodres, R. Schell and C. Cotton for the quick turnaround they have given the many drafts of chapters I have given them in the last hectic weeks of thesis preparation, and for their patience and understanding.

I have to thank my second reader from Computer Science Department Professor R. Schell. Professor R. Schell during this research represented for me a never ending source of new ideas and quick solutions. He provided me with quick and correct solutions to many seemingly unsolvable problems, and I greatly appreciate the many hours of his time, I spent, helping me to clarify my work.

I have to especially thank Professor T. Tao who provided me with the environment and all the necessary tools to carry out this research.

I have to thank Professor U. Kodres for his support and help in the implementation of the hardware interrupt structure needed to provide the feature of preemptive scheduling.

Finally, I would like to thank my wife, Marika and my children Anastasia and Helen for their help, patience and understanding during the last two weeks of thesis preparation.

I. INTRODUCTION

The topic of this thesis is the detailed design and implementation of the kernel of a real-time, distributed operating system for a multiple microcomputer system.

A. MOTIVATION

In the Electro-Optics Signal Processing Laboratory at the Naval Postgraduate School, research is currently being conducted in the area of "smart sensor image processing". Specifically image processing for long distance missile detection, high-altitude surveillance and target acquisition for tactical missiles is the topic presently being investigated.

The smart sensor platform will require on-board data processing of large quantities of collected image data.

To provide the required "computing power" for a high input data rate which processes that data in "real time", a multiple microcomputer system is being developed capable of performing concurrent asynchronous computations.

A large image processing program can be partitioned into small interactive parts. These will be dynamically assigned to the microcomputers available in the system for concurrent "parallel" and "pipeline" processing.

If properly designed and executed, the concurrent computing will both increase the throughput and decrease the execution time.

To facilitate the dynamic assignment of the partitioned processes of a program for effective computations by a multiple microcomputer system, a real-time distributed operating system is needed and this is the topic of the present thesis.

B. DISCUSSION

The processing power of microprocessors is increasing. If this power can be effectively coordinated by an operating system, it would provide a more affordable and powerful product.

The application of contemporary microprocessor technology to the design of large-scale multiple processor systems offers many potential benefits. For example, the "cost" of high-power computer systems could be reduced drastically, and "fault tolerance" in critical real-time systems could be improved. Designing such systems presents many formidable problems that have not been solved by the single processor systems available today.

The multi-microprocessor systems in use today suffer performance degradation as more processors are added to the system. Sophisticated interconnections among processors and memories are needed to reduce this problem.

Despite the rapidly expanding capabilities of modern microcomputer systems, they still are limited by the relatively slow execution speeds of their microprocessors,

for many real-time military applications. These systems generally do not provide the power and flexibility required to address complex and demanding applications. One such area is that of "real-time digital image processing". This is a particularly demanding application area, characterized by the requirement to apply significant "processing power" to a high input data rate.

An answer to the inadequacies of the single microcomputer is to provide for miltiple microcomputer systems. Such systems could provide the processing power necessary to handle those applications, which are presently addressed by mini-computers and mainframe systems. However, most of today's microcomputer operating systems deal only witha single processor and cannot adequately manage multiple processors.

The integration of large numbers of relatively inexpensive microcomputers into powerful computer systems has been the subject of intensive research in universities and industry for several years.

The primary thrust of this thesis is towards a general architecture which can be applied to hardware systems, that are commercially available today (this project is currently using the INTEL general purpose 16-bit 8086 Microprocessor), with some custom-developed hardware for intercommunication network and control.

C. BACKGROUND

The system software design uses the MULTICS [9] concepts of segmentation and "per process stack", in conjunction with Reed's [15] design of virtual processors, and Reed and Kanodia [10] eventcount synchronization mechanism.

The basic microcomputer operating system design was developed by O'Connell and Richardson [7] and is based on the structure of a hierarchical kernel, where security kernel technology was used. O'Connell and Richardson first developed a flexible operating system design that is fundamentally configuration independent and adaptable to a spectrum of systems.

J. Wasson [8], in his thesis defined the detailed kernel design of one member of the above family, a modified real-time subset, tailored to "real-time image processing" and applied to the INTEL 16-bit general purpose 8086 Microprocessor.

The objective of this thesis is to complete the above design and also to write a detailed code implementation.

The result is a layered loop free operating system which is both small and easy to analyze.

The system supports miltiple asynchronous processes, using the concept of "two-level traffic control", to accomplish "processor multiplexing" amongst a greater number of eligible processes. This dual-level "processor multiplexing" design allows the system to treat the two primary scheduling

decisions, viz., the scheduling of processes and the management of processors, at two separate levels of abstraction.

The kernel comprises a complete, albeit primitive, operating system providing support for a large number of asynchronous processes.

The kernel manages all physical processor resources, and provides scheduling and interprocess communication and synchronization and also provides the user with an execution environment which is relatively free from concern about the underlying hardware configuration. The system is capable of performing in a real-time environment through the use of "preemptive scheduling", to ensure expeditious handling of time-critical processing requirements.

D. STRUCTURE OF THE THESIS

Chapter I presented a general discussion and the thesis' background.

Chapter II, describes the overall design philosophy of the operating system, its functional requirements, how multiple processes communicate and synchronize their tasks, and finally how these processes are multiplexed on a smaller set of processors.

Chapter III describes the hardware architecture of the multiprocessor system. The INTEL 8086 Microprocessor was chosen for this implementation.

Chapter IV describes the details of the system design.

Chapter V presents conclusions and observations that resulted from this effort and also suggestions for further research.

II. FUNDAMENTAL DESIGN CONCEPTS

A. DESIGN PHILOSOPHY

The kernel primitives which provide multiprogramming processor management and process management, form one member of the family of operating systems designed by O'Connell and Richardson [7]. This member is a modified real-time subset. The modification consists of the inclusion of a more general synchronization mechanism, eventcounts and sequencers described by Reed and Kanodia [10], which replace the more traditional Signal/Wait and Block/Wakeup used in the original design.

Before presenting the details of this operating system, the high level design and the detailed "working implementation" of the system, it is useful to investigate the general design methodology applied to the development of this operating system.

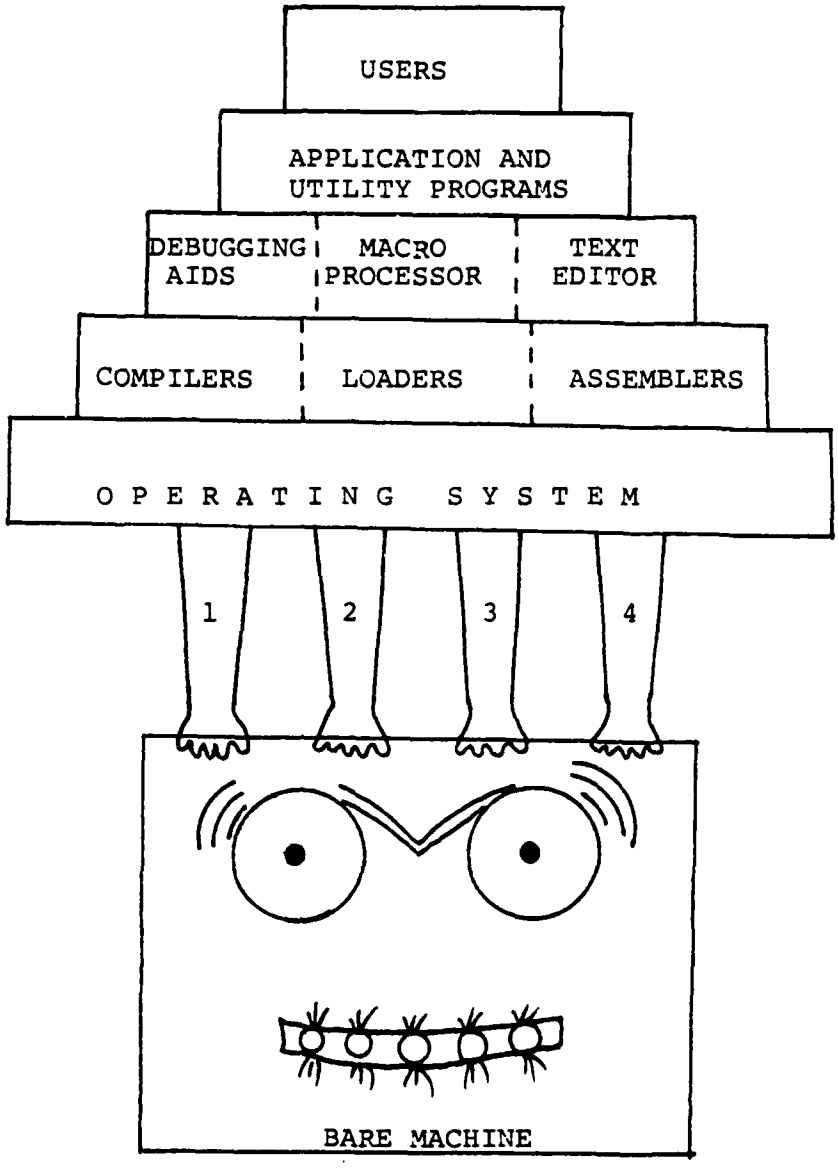
Multiple processor systems are intrinsically more complex than the familiar uniprocessor. Their complexity has proven to be the major barrier to realize the full potential of the inherent parallelism available in such a system.

One of the most important components of any computer system is the operating system. The operating system manages the system's resources. Thus system performance is critically dependent upon its effectiveness.

We can say that basically two issues confront the operating system designer. First, he must provide system functions that support the services requested by the user. These functional requirements affect the logical design of the system. Second, he must address issues of cost and performance. Cost and other management considerations will not be addressed here. Performance issues concern the management of physical resources and has to do with the computational speed, and also system attributes such as the ease of programming, efficiency, correct operation, etc.

There is a considerable amount of literature devoted to the development of the functional design of operating systems. Dijkstra [12] has described a technique for reducing the complexity of the design by allocating operating system activities to a number of cooperating processes. Process structure is simplified in turn, by defining its functions in levels of increasing abstraction, and by applying the principles of structured programming.

Madnick and Donovan [13] have described an operating system as a hierarchical extended machine. Program modules are added to the system to provide many extended instructions, in addition to the hardware instructions available on the bare machine. In complex systems, one extended machine may be constructed upon another to form a system composed of levels of abstraction (virtual machines). Figure 1 from Reference [13], presents the general idea of that hierarchical



1. Memory Management
2. Processor Management
3. Device Management
4. Information Management

FIGURE 1. RELATION OF OPERATING SYSTEM TO COMPUTER HARDWARE

extended machine and levels of abstraction built on the bare machine.

Saltzer [14] and Reed [10, 15] have discussed the advantages of resource virtualization and have described useful interprocess communication and synchronization mechanisms. The general design strategies presented in this thesis will aid the operating system designer in developing system functions in a clean, logical, verifiable design.

Finally, adequate performance can only be assured if the behavior of the system is well understood and this in turn imposes a strict requirement for simplicity.

In this design, the requirement for simplicity is satisfied by utilizing a model based on the notion of multiple asychronous processes with segmented address spaces. This is the central unifying concept which provides a straightforward view of both static and dynamic system behavior [4]. The principles of structured system design are also applied to logically organize the operating system into a hierarchically structured set of easily understood modules whose interactions are clearly specified and strictly enforced.

The result is a modular, layered operating system which makes it easier to ensure correct operation and provides better opportunity for improving performance through tuning. Finally, because the system is small, less memory is used for

operating system code and less processor time is spent in its execution.

The operating system design is logically organized into a hierarchy that separates the user application processes from the kernel. This modular, layered design lends itself to "dynamic reconfiguration" where processes can be relocated among physical processors [19]. Additionally, the system initialization technique proposed by Anderson [19] provides a basis for an automatic recovery mechanism that will initialize the system on a new physical configuration after the detection of faulty system components.

B. FUNCTIONAL REQUIREMENTS

The functional requirements defined below support the specific design goals of the system and provide features desirable in any operating system, such as: a logical structure, fault tolerance and efficiency of operation. Functional requirements define services that must be provided to support the user's environment.

1. Process structure

By dividing a job into asynchronous parts and executing these parts as separate entities, significant benefits can be realized. Within a single processor system the partitioning into asynchronous parts provides the "only" design simplicity. But in a multi-processor system, the partitioning into asynchronous parts is essential, if the

"parallel and pipeline processing" potential of the system is going to be used.

2. Definition of a process. Process organization

The abstract idea of a process has been defined in several ways. A simple one offered by J. Saltzer is the following:

> "a process is a program in execution on a processor." [14]

A process is the sequence of actions taken by some processor. In other words, it can be viewed as the past, present ad future "history" of the state of the processor. The notion of a process provides a complete description of all instructions executed and all memory locations referenced during the performance of a task.

Considering the above definition, it becomes clear that there are two elements which together completely characterize and define a given process. These are the process' "address space" and the "execution point."

The address space is the set of memory locations that could be accessed during process execution. The execution point is the state of the processor at a given instant during process execution (and is characterized by the contents of certain processor registers).

In the abstract view, an address space is defined by a collection of discrete points, each representing a memory word. The process is described by the path traced through

this address space from process creation to its destruction. In Figure 2 the main path, shown by a heavy black line, traces the process execution point as it moves from one instruction (i.e., memory word) to another during process execution. The branches from this execution point path represent data references.

The concept of a process has proven to be a fundamental and powerful one in the organization of computer systems. By designing a system as a collection of cooperating processes, system complexity can be greatly reduced. This is because the asynchronous nature of the system can be structured logically by representing each independent sequential task as a process and by providing interprocess synchronization and communication mechanisms to prevent "race" and "deadlock" situations during process interactions.

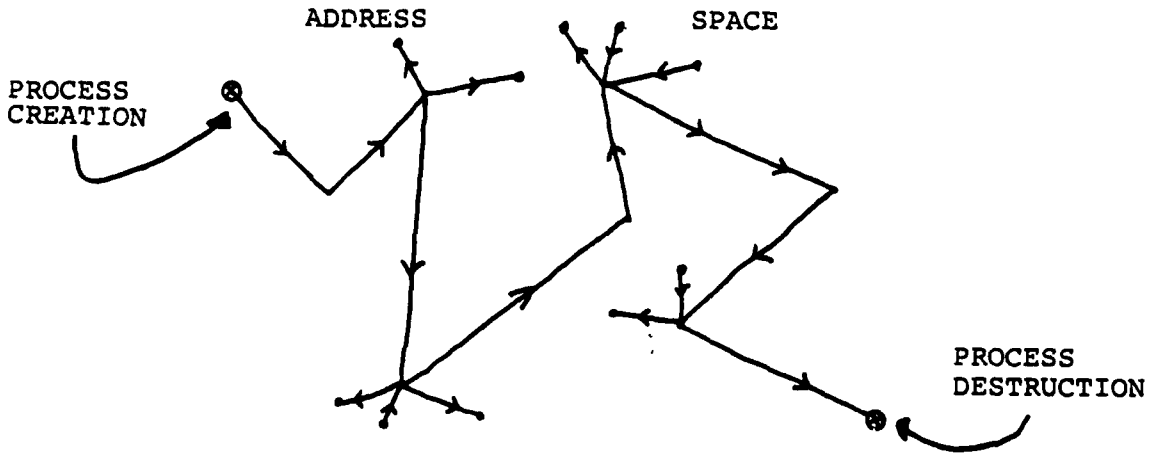


FIGURE 2. PROCESS HISTORY

Several advantages result from using this process oriented design. As a tool for dealing with the asynchronous nature of system operation, processes provide a simple, logical, high-level structure for the design. Since each process is confined to a specific address space, tasks are isolated from one another and system fault tolerance can be improved.

Each process is assigned a unique identifier and is an explicit entity that requires management.

In a "distributed" operating system, those portions of the operating system that are logically part of the sequential flow of control (viz., locus of execution) are within the address space of each user process. This is made possible by dividing the operating system into procedures that are called like any other application procedure.

It should be noted that in a distributed operating system there is no "master" assigning processes to processors. Rather, each running process "gives up" its processor to the next process that is ready to run.

The address space of a process we can say, provides a container for the process which isolates it from any other process. This eliminates the possibility of inter-process interference simply because processes are unable to "escape" the confines of their defined address spaces.

However, this is rather restrictive in that processes which are totally ignorant of each other have no hope of co-operating towards the accomplishment of some greater goal. In order to mediate this constraint, one desires to allow some restricted (controlled) form of address space overlap, (viz., sharing), such that co-operation is allowed while still retaining the benefits of protection offered by isolation. Sharing requires some way of distinguishing the shared portions of the address space. This is greatly facilitated by introducing the notion of memory segmentation.

Finally, to distinguish between a process and a processor (physical or virtual), we can say that the major difference is that a processor is an "actor", while a process is a sequence of "actions" taken by that actor. A process results from the actions of a processor.

3. Virtual Memory and Segmentation

In many memory handling schemes, processes cannot run unless the entire address space is loaded in primary memory. This may require a large main memory or it may restrict the size of the address space. An alternative plan requires an operating system which manages primary and secondary memory to create the "illusion" of a memory which is larger than system's primary memory. Since the larger memory is only an illusion,it is often called "virtual" storage.

Virtual memory is used to implement the concept of a "per process" address space. In Multics [16] each process is

provided with its own virtual memory for an address space. These virtual memories are completely independent of one another.

A virtual memory (the address space of a process) is composed of a set of segments. A segment is a logical collection of information (e.g., procedure, data structure, file, etc.) and is the basic logical object of this design. Segments are distinct "variable size" memory objects containing a sequence of words with conventional linear addresses. Associated with a segment is a set of logical attributes used to uniquely identify the segment and to control access to it.

In specifying the set of segments that comprise a virtual memory, one may include segments that are also part of "other" virtual memories as well. So in addition, segmentation supports "information sharing" since a segment may belong to more than one address space. Segmentation provides a means of associating logical attributes and labels with each segment, such as, access class, domain, etc. Thus, segments can be shared in a controlled manner to provide for inter-process communication and synchronization.

By using segmentation to provide a virtual memory environment, the user is presented with a configuration independent system in that he "sees" a process address space that he can consider "his own" and is not dependent on the assignment of physical addresses.

a. Addressing in a Segmented System

Addressing in a segmented memory system is "two-dimensional", That is, a complete address consists of two parts. The first is the "segment number", This identifies the particular segment of interest. One attribute of the segment is the physical address of the segment's base. Thus the segment can be located anywhere in physical memory just by changing this base address. The second dimension of the address is an "offset" relative to the segment's base (the beginning of the segment). This serves to access specific locations within the segment.

This two-dimensional addressing "frees" information from dependence on a particular memory location by making it arbitrarily "relocatable",

Figure 3 illustrates the two-dimensional nature of the segment address. The descriptor segment provides a list of descriptors for all segments in a process address space. As previously mentioned, one attribute of the segment, given by the segment descriptor, is the "physical" address of the segment's base. Then the second dimension needed to access a specific memory word within this segment is given as an "offset" from the segment's base, (SEG # n, OFFSET), e.g., (1st dimension, 2nd dimension). So, in segmented addressing, each address is characterized by an ordered pair of numbers (1st dimension, 2nd dimension).

Because of the similarities in address mapping hardware, very often the distinction between paging and segmentation is confused. To distinguish between page and segment, we emphasize the conceptual differences here. The major difference is that a segment is a "logical" unit of information "visible" to the user's program and is of "arbitrary size", A page is a "physical" unit of information strictly used for memory management "invisible" to the user's program and is of a "fixed size", In this design only segmentation is supported by both the hardware and software.

Segmented memory management can offer several advantages. It can: control fragmentation; facilitate shared segments (data areas and procedures); and also for future development in this system can provide dynamic linking and

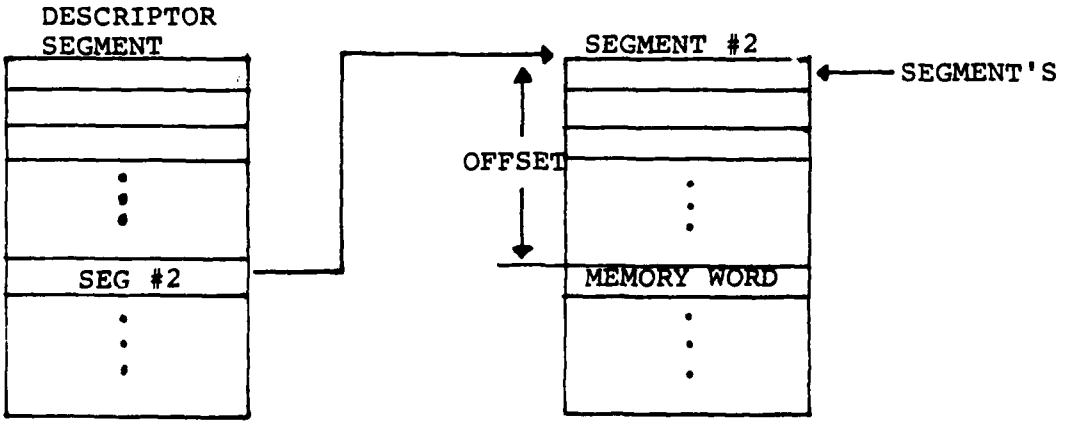


FIGURE 3. SEGMENTED ADDRESSING

loading, controlled access, dynamically growing and shrinking segments. More details about segmentation in the present design will be discussed in the next chapter.

4. Abstraction - Abstract types

"Abstraction" provides a method for reducing problem complexity by applying a general solution to a collection of specific cases [17]. Structured programming provides a tool for creating abstraction in software design.

An "abstract type" is a class of objects in the system, for which there is a defined set of operations. The difference between an abstract type and the "classic" notion of type, is that the user of an abstract type need not know the representation of the object or the algorithms used to implement operations defined on the type, and furthermore, the only operations allowed to be performed on the object are specified by the definition of the type.

The concept of abstract type is quite attractive for the structuring of large systems. The result is the kind of structuring prescribed by Parnas' "information hiding principle" [20], for decomposing a system into modules. Further, abstract types fit naturally into the structure of an operating system since a major task of an operating system is to multiplex a set of physical resources to produce a set of virtual resources that can be viewed as objects of abstract type. For example, this is exactly what happens in processor multiplexing (see paragraph C).

An abstract type consists of a set of objects and a set of operations. For example, a word in virtual memory is an abstract type. Two operations that can be carried out by instructions in user processes are read-word, which obtains the content of a word named by a particular virtual memory address, and write-word, which takes a bit string and stores it in the object specified by a particular virtual memory address. Processors, both real and virtual, also can be viewed as objects of abstract type.

The abstract type idea clearly furnishes a useful way to view the virtual objects seen at an operating system, but for the design of an operating system the abstract idea is equally important in structuring the internal implementation of the system.

By strictly applying two special rules in addition to the general principles of structured programming, a structure consisting of levels of increasing abstraction can be constructured.

First, calls cannot be made outward toward higher levels of abstraction. This frees lower levels from a dependence on higher levels by creating a loop-free structure and results in a design which is capable of having subsets.

Second, calls to lower levels must be made through specific entry points or gates. Each level of abstraction creates a virtual (hierarchical) machine [13]. The gate to each level provides a set of instructions created for that

virtual machine. Thus, higher levels may use the resources of lower levels only by applying the instruction set of a lower level machine. Once a level of abstraction has been created, the details of its implementation are no longer an issue. Instead users see layers of virtual machines, each defined by its extended instruction set.

For this particular design when the rules of abstraction are applied to level Ø, the physical resources of the system, these resources are "virtualized". Thus the first level of abstraction creates "virtual processors", "virtual memory", and "virtual devices" from the system's hardware. At each higher level the detail of the design is reduced. The gate at the boundary between the highest level of the kernel and the lowest level of the supervisor provides a mechanism for isolating the kernel as well as ensuring that each memory access is via kernel software. This mechanism has been implemented in the system by a ring-crossing mechanism called the Gatekeeper (or Gate).

5. Protection Domains - Levels of Abstraction

a. Protection Domains

The implementation of this operating system has not considered the "internal security" of the system but in the design there are all the ingredients for future extensions in this direction.

An essential requirement [22] of internal security is that the security kernel be isolated from other elements

of the system. This can be accomplished by the construction of protection domains. Protection domains are used to arrange process address spaces into rings of different privilege. This arrangement is a hierarchical structure in which the most priviledged domain is the innermost ring. The structure essentially divides the address space into levels of abstraction with strictly enforced gates at the ring boundaries (Figure 4).

The protection provided by the ring structure is not a security policy (security protection is implemented by a lattice structure). It is, however, a mechanism to enforce the hierarchy of the virtual machine by creating a priviledged kernel ring within the supervisor ring.

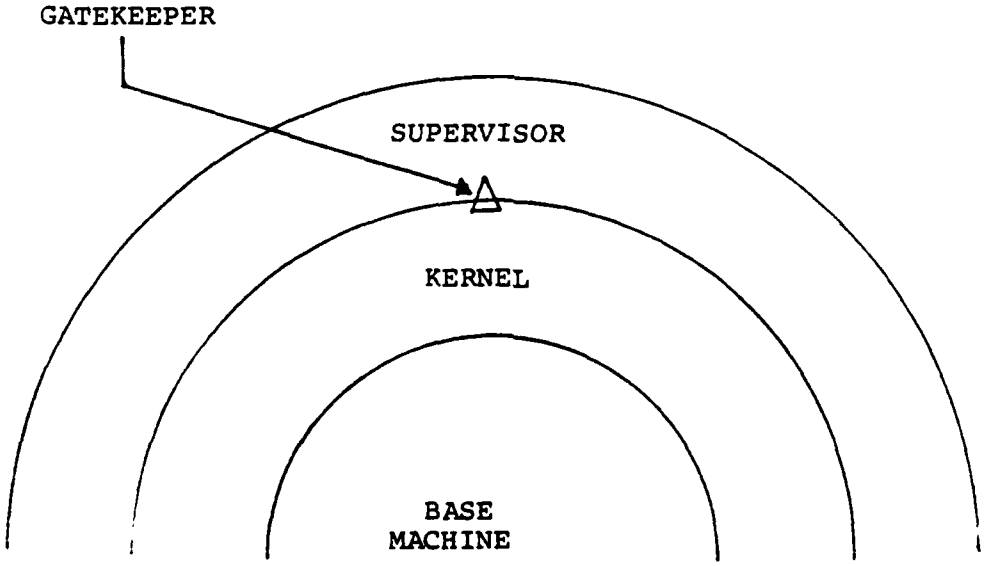


FIGURE 4. PROTECTION RINGS

In this implementation to protect kernel procedures from the user, the process' address space is divided into two hierarchical domains, "user domain" and "kernel domain". The kernel domain is the most priviledged. Only the kernel executes in this domain. The user domain is less priviledged and is separated from the kernel domain to protect the user from inadvertently causing problems to the operating system services. These two domains are generated by software since there is no hardware support.

b. Levels of Abstraction

Abstraction is a way of avoiding complexity and a tool by which a finite piece of reasoning can cover a myriad of cases [17]. The purpose of abstracting is not to be vague, but to create a semantic level in which one can be absolutely precise.

Levels of abstraction have been demonstrated to be a powerful design methodology for complex systems. The use of levels of abstraction in general leads to a better design, with greater clarity and fewer errors.

A level is defined not only by the abstraction that it supports (for example, a segmented virtual memory) but also by the resources employed to realize that abstraction. Lower levels (closer to the hardware) are not aware of the abstractions, or resources of lower levels only by appealing to the functions of the lower levels. This pair of restrictions reduces the number of interactions among parts of a system and makes them more explicit.

Each level of abstraction creates a virtual machine environment. Programs above some level do not need to know how the virtual machine of that level is implemented. For example, if a level of abstraction creates sequential processes, and multiplexes one or more hardware processors among them, then at higher levels the number of physical processors in the system is not important. This way, in present implementation, since the processes are assigned virtual processors (and not physical), there is no effect on the user when real processors are added or deleted (except, of course, for the change in performance). Adding and deleting processors will have particular interest when "fault tolerance" and "fault correction" are added to the attributes of the operating system.

On the present implementation, the operating system is structured as a hierarchy of the levels of abstraction shown in Figure 5.

C. PROCESSOR MULTIPLEXING

1. Definition of a Processor

The basic function of a processor is to perform a sequence of operations on objects in its environment. The environment of a processor is a set of objects. For example, the environment of a physical processor is that portion of memory that it can access through its address mapping hardware. Typically the environment is specified by an object

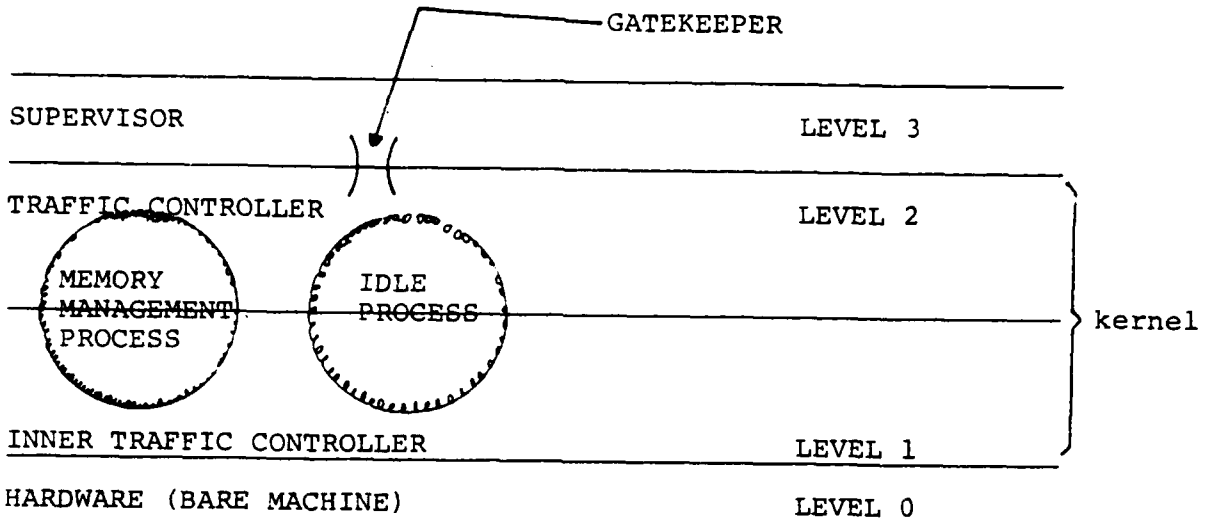


FIGURE 5. LEVELS OF ABSTRACTION

such as the "descriptor segment" (in MULTICS) which in turn names another object.

A processor has internal memory, called its state, that it uses to pass information from one operation to the next. The processor determines the next operation to perform by interpreting an instruction found in the processor's environment by an instruction pointer that is part of the processor state. Also included in the processor state is the name of the current domain in which the processor is executing.

Each operation performed may modify the contents of the processor's internal memory. In particular, it changes

the instruction pointer to select the next instruction to be interpreted.

As an object of abstract type, a processor may be part of the environment of other processors. The operations that can be performed on a processor object are: loading a new state into the processor, extracting the current state from the processor, causing the processor to run, causing the processor to stop, etc.

A processor can be a physical object such as the INTEL 8086, 16-bit general-purpose microprocessor used to implement this design. In this case, the processor registers comprise the state of the processor. The environment of the processor includes all of the primary memory that is accessible through the processor's descriptor segment. The descriptor segment in this design is related to the four hardware segment registers (CS, DS, SS and ES). Details are discussed in the next chapter.

On the other hand, a virtual processor which has no direct hardware manifestation, is a simulation of a physical processor achieved by using physical processors to interpret the instructions to be executed by the virtual processor. The virtual processor idea is discussed in the following paragraph 3.

2. Definition of Processor Multiplexing

Multiplexing can be defined as the use of a single resource for different purposes at different times. For

example, the physical bus lines can be used both for addresses and data during different times of a machine cycle.

Processor multiplexing is a technique for sharing scarce processor resources among a number of processes. The ability to multiplex processors efficiently provides a mechanism for the virtualization of these physical processors by simulating the existence of a larger number of virtual processors. This technique is widely used in conventional uniprocessor systems where it is called multiprogramming. It seeks to maximize the use of the available hardware by automating control of process loading and execution. It also greatly increases the flexibility of a system allowing it to be effective in more complex and demanding applications.

J. H. Saltzer [14] presented one of the fundamental works on the subject of processor multiplexing.

3. Processor Virtualization

The first levels of abstraction, above system hardware, creates virtual representations of physical resources (virtual processors, virtual memory). Since upper levels of the design operate on these virtual processors rather than on physical processors, most of the design (i.e., everything above virtualization level) is independent of the physical configuration of the system. This means that by providing the virtual to real processor binding in the kernel of the operating system and since the processes are assigned virtual processors (and not real processors), there is no effect

on the user when real processors are added or deleted in the system (except, of course, for the change in performance).

The physical processor resources (those hardware devices that execute machine instructions) are virtualized by creating abstract processors called virtual processors.

Processor multiplexing can be defined also as a simulation of a number of distinct virtual processors by a smaller number of real processors.

a. Virtual Processors

A virtual processor is a data structure that contains a complete description of a process in execution on a physical processor, at a given instant. This description is contained in the process execution point. The address space of the process must be accessible to the virtual processor when it is "loaded" on ("bound" to) a CPU. To provide a useful virtualization capability, the CPU must have the ability to efficiently multiplex process execution points and address spaces (i.e., it must support multiprogramming).

Virtual processors are simulations of processors. They can be viewed in essentially the same way as physical processors, in that they execute the same instructions. However, the instruction set of a virtual processor has been expanded to include some instructions which the physical processors do not directly have. These include "instructions" to "load" a process, certain operation called interprocess

communication and synchronization primitives, system service calls, etc.

For example, the AWAIT operation is not an operation that requires real processor resources, it is rather an operation that inhibits use of real processor resources by the virtual processor.

Virtual processors exist only as "abstract" processors represented by a data structure. They are used as the vehicle for the control and manipulation of processor resources.

Each of the virtual processors executes a sequence of operations in time. These sequences are actually performed by the real processors. Successive operations of the same virtual processor may be separated by a gap of time, during which operations of another virtual processor are being executed by the real processors. Figure 6 shows how the operations of three virtual processors might be mapped into the operation sequence of one real processor

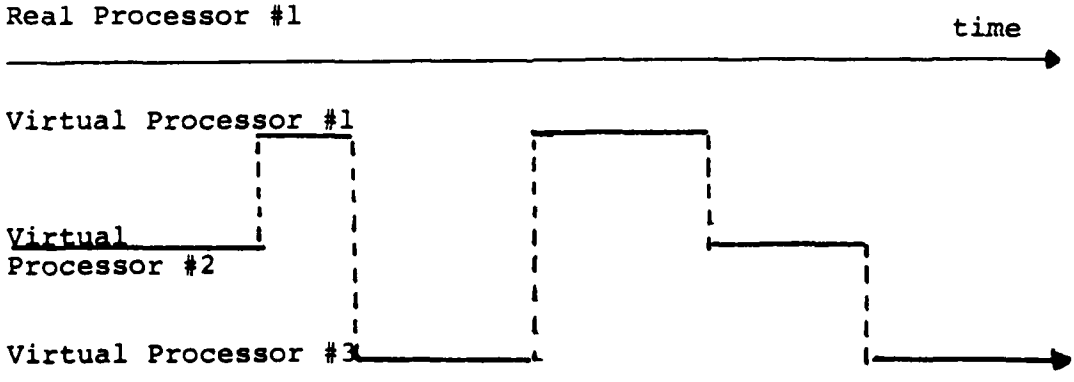


FIGURE 6. MULTIPLEXING A REAL PROCESSOR

To define a term used frequently in this thesis, a virtual processor being simulated by a real processor is "bound" to that real processor whenever its process is being executed by the real processor. Thus, Virtual Processor #2 in Figure 6 is bound to Real Processor #1 during the first time interval.

Processor multiplexing also requires a policy of scheduling. Given a number of virtual processors to which a real processor may be bound at any one time, the real processor can execute only one virtual processor. The choice of the processor to run is made by some algorithm called virtual processor scheduler. This algorithm receives as input the set of virtual processors belonging to this real processor and chooses which one is to run (be bound and execute).

4. Multiprogramming

Multiprogramming is used to improve system efficiency and to create a virtual environment which frees the remainder of the operating system from a dependence on the physical processor configuration. Processor management provides a means of coordinating the interaction of the asynchronous processes which comprise the system. This implementation employs a processor multiplexing technique for a distributed kernel and provides a virtual interrupt mechanism. The modular hierarchical structure of the software is "loop-free" to support future system expansion to higher level functions.

The clean, logical, process-oriented structure of the system offers other benefits as well, including possible inclusion of rault tolerance, resource configuration independence, and efficiency.

In a system where there are more processes than processors, there must exist a means of switching processors from process to process. For example, reasons for switching processes are: current process completes, current process is blocked, a higher priority process is ready to run, etc.

Whatever the reason for switching, there are certain tasks that must be done in performing the switch. First, save the address space and current execution point of the old process. Secondly, load the address space and the execution point of the new process.

5. Multiprocessing

The process structure provides the essentials for parallel processing. That is the support for a set of asynchronous processes which can communicate with each other. Parallel processing does not require a multiprocessor environment. However, in a multiprocessor environment, parallel processing can provide faster completion of a job.

Whenever a job depends on a mixture of asynchronous and synchronous tasks and time is a factor, then concurrent processing is a possible solution to get the job done in the specified amount of time. Using several processors working on the same job and each of them doing separate tasks, the overall time required to to this job can be reduced (job has been structured into explicit processes).

The above discussion provides some of the major reasons why this system was designed to support parallel processing on multiple processors.

6. Two-Level Processor Multiplexing

In this design there are two levels of processor multiplexing. The design in two levels arose from the existence of multiple physical processors. Each of the levels addresses a distinct requirement. One level supports virtual processor management, that is, the provision of inter-process communication and synchronization. The other supports the management of physical resources by the operating system. The first one addresses the multiplexing of virtual processors among processes and is the "Traffic Controller". The other addresses the multiplexing of physical processors among virtual processors and is the "Inner Traffic Controller".

a. The Traffic Controller

The Traffic Controller represents the upper level of processor multiplexing (Level 2) and provides the mechanism for multiplexing virtual processors among processes. Thus it is responsible for inter-process synchronization and communication.

As an example, consider that a Process A wishes to synchronize its actions with another Process, B, such that Process B has to complete some task before A can continue execution. Thus A will execute to the point where

it cannot proceed further and then wishes to signal B. When Process B has finished that task it must notify Process A of its completion so that Process A may then proceed.

This inter-process synchronization and communication is handled at the level of the Traffic Controller. In the above example, when Process A discovered that it could not proceed further, it "gave away" its virtual processor to another process that could be run. In this way the Traffic Controller suspended the execution of Process A and a new process was bound to its virtual processor. In the same way, when B completes (viz., it has no more work to perform) it will also give its virtual processor away.

b. The Inner Traffic Controller

The Inner Traffic Controller comprises the lower level of processor multiplexing (Level 1) and provides the second set of multiplexing functions. It multiplexes physical processors among a fixed set of virtual processors. In particular, the system's interrupt structure is managed by the Inner Traffic Controller.

If a user process calls upon some system service, such as disk I/O or I/O for a real-time sensor, it must wait for that service to be completed before it can proceed. The performance of a system service is considered to be part of the requesting processes. However, the service may actually be supported by another virtual processor. To control this interaction, the Inner Traffic Controller provides the

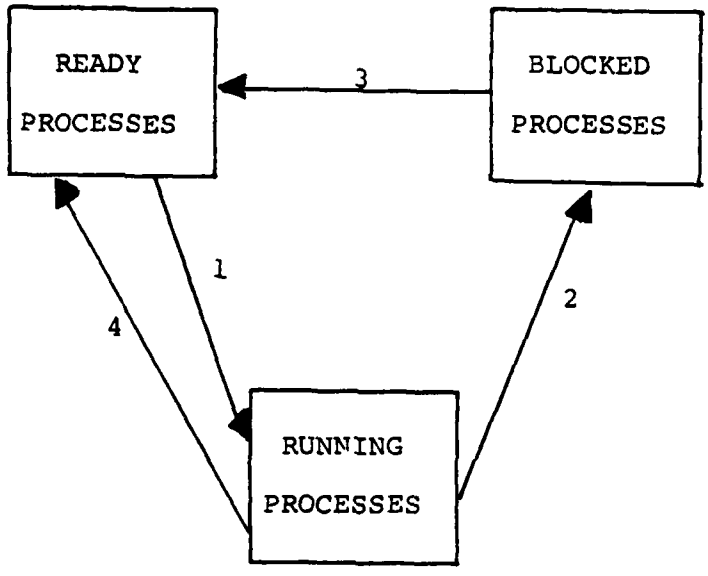
required inter-virtual processor synchronization and communication mechanism. In particular, a physical system interrupt is directly transformed into a synchronization signal to a waiting virtual processor. This structure is particularly important for the support of real-time processing, and note that it is completely distinct from inter-process synchronization and communication described in paragraph 6a.

c. Processor Multiplexing Strategy

1. Process State Transitions

Figure 7 illustrates the state transitions of a set of processes as a virtual processor is multiplexed among them. Some eligible process (one which is in the ready state) is scheduled to run and is "bound" to the virtual processor. At this time, the process makes the transition to the running state. As far as the process is concerned, once it enters the running state, it is executing.

At some point in its execution, the process may desire to block itself or signal another process. (For example, when Process A is at that execution point and needs data computed by another Process, B.) In that case, it will block itself (will enter the blocked state) and will "give up" the virtual processor to which it is presently bound and will be out of "contention" for processor resources. It will remain in the blocked state until some other process will signal it. (In the above example, when Process B has computed the needed data for Process A.) Then this process will make



STATES: READY, RUNNING, BLOCKED

TRANSITION 1: Decided by the Traffic Controller scheduler (higher priority ready process)

TRANSITION 2: After a Traffic Controller AWAIT operation.

TRANSITION 3: After a Traffic Controller ADVANCE operation.

TRANSITION 4: After a Traffic Controller ADVANCE operation (and effect of preemptive scheduling).

FIGURE 7. PROCESS STATE TRANSITIONS

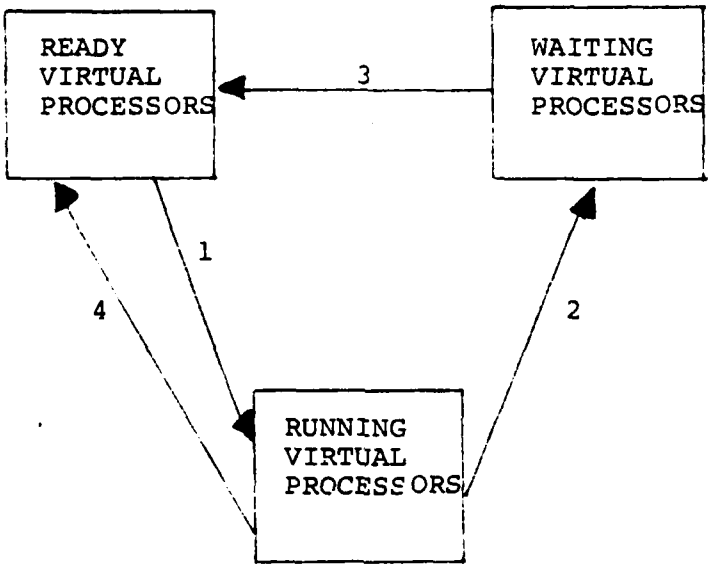
the transition back to the ready state. If the process signals other processes, it will make a transition from the running state back to the ready state from which it may be scheduled to run again. In doing so, it allows the Traffic Controller to possibly give the virtual processor to some other higher priority process which may be ready to run.

The mechanisms which decide and permit these transitions are the Traffic Controller Scheduler and the Traffic Controller inter-process synchronization and communication primitives AWAIT, ADVANCE. Their details will be discussed in Chapter IV.

b. Virtual Processor State Transitions

Figure 8 illustrates the state transitions made by the virtual processors as a physical processor is multiplexed. This diagram is very similar to that of Figure 7. However, these transitions are not directly observable by processes except in the differences of their execution times, as virtual processor state transitions result from the management of physical resources by the operating system.

A running virtual processor can make a transition to the waiting state or to the ready state. The transition to the waiting state occurs when a virtual processor must wait for the completion of some system service (analogous to the blocking of Process A in the example given in paragraph a). The transition from running state back to ready state occurs



STATES: READY, RUNNING, WAITING

TRANSITION 1: Decided by the Inner Traffic Controller scheduler.

TRANSITION 2: After an Inner Traffic Controller AWAIT operation.

TRANSITION 3: After an Inner Traffic Controller ADVANCE operation.

TRANSITION 4: After an Inner Traffic Controller ADVANCE operation.

FIGURE 8. VIRTUAL PROCESSOR STATE TRANSITIONS

when the running virtual processor signals other virtual processors. It will allow the Inner Traffic Controller to possibly run another higher priority virtual processor. While in the waiting state, the virtual processor is out of "contention" for processor resources until another virtual processor signals it to continue. While in the ready state, the virtual processor is in contention for processor resources and so may be scheduled to run on the physical processor.

The mechanisms which decide and permit these transitions are the Inner Traffic Controller scheduler and the Inner Traffic Controller inter-virtual processor synchronization and communication primitives AWAIT, ADVANCE. Their details will be discussed in Chapter IV.

D. COMMUNICATION AND SYNCHRONIZATION

For concurrent processing, a job that is composed of sequential and non-sequential tasks is explicitly divided into an appropriate structure of processes that can run concurrently. Inter-process communication and synchronization are necessary for concurrent processing.

It is the responsibility of the operating system to provide mechanisms for communication between cooperating processes. There are two different kinds of communication that processes must be able to achieve.

There must exist a way for processes to exchange data in some way. This mode of communication is called inter-process communication.

There must also exist a way for processes to wait for data prepared by other processes, and for processes that prepare such data, to signal that this data is available. This interaction is different than communication of data, and is called inter-process synchronization. Together these two modes are called inter-process communication and synchronization.

The actual coordination for the exchange of data between processes is realized by the use of "shared writable" segments.

Therefore, to utilize the parallelism and pipelining afforded by multiple processors, a mechanism is required for inter-process communication and synchronization. It is used for controlling the execution of processes and coordinating the sharing of data.

The most widely used synchronization primitives are Dijkstra's semaphores [11] or Saltzer's Block and Wakeup [14] that were used in O'Connell and Richardson's original design [7]. However, the design decision was made to use a different mechanism which provides automatic "broadcasting", supports "parallel signalling" and addresses the questions of "confinement" (or * property) in a secure system. This is the synchronization mechanism based on the design of eventcounts and sequencers of Reed and Kanodia [10].

The synchronization between processes is supported by the AWAIT and ADVANCE, that are the kernel calls to the Traffic Controller level. The Traffic Controller is the

kernel module that manages processes and supports scheduling for user processes by multiplexing the user processes into a limited (fixed) number of virtual processors.

AWAIT and ADVANCE are primitives of the Traffic Controller. These primitives can be used to provide simple cooperation, such as mutual exclusion or complex interactions, when required by the application. How the user's procedures invoke the AWAIT and ADVANCE primitives depends, of course, on the actual process structure. (Examples will be given in Appendix A).

A process can only block itself (using AWAIT) and cannot block another process. The AWAIT sets the "calling process" that invoked AWAIT in the blocked or ready state and then the Traffic Controller Scheduler schedules another ready process to run, the highest priority ready process.

The ADVANCE is used to provide asynchronous processes with a synchronization signal. The ADVANCE takes as parameter the name of the associated eventcount. It advances the value of that eventcount by one. This incrementation of the evencount value is "broadcast" to all the processes waiting for that event. Then a check is made to determine if the awaited eventcount value (for the processes waiting that event) is smaller or equal to the current value of the eventcount. If this is the case, then these previously blocked processes will awake and resume the ready state. Otherwise they will remain blocked. Then a check is made to find out if the

currently running process is of lower priority than the other ready processes (after ADVANCE operation). If that is the case, the ADVANCE will send to the virtual processor (which is running this lower priority process) a "pre-empt interrupt". Finally the scheduler will select the highest priority ready process to run. So, we see that the ADVANCE is also responsible for operating the "pre-emption mechanism".

The above describes roughly the idea of AWAIT and ADVANCE primitives, which are very close to the BLOCK and WAKEUP described in O'Connell and Richardson's thesis [7]. More details and the whole operation of eventcounting will be discussed in Chapter IV.

Another system level concerned with synchronization is the Inner Traffic Controller. This level manages the physical (real) processors to create the virtual processors, that are in turn managed by the Traffic Controller.

The Inner Traffic Controller provides the interface and does the multiplexing among the physical (real) and virtual processors. Each physical processor has associated with it several (a fixed number) of virtual processors. Some of these virtual processors are mutliplexed in turn by the Traffic Controller among user processes. Each system process is assigned (dedicated to) a virtual processor. In the current implementation there are two such processes, and these will be discussed in Chapter IV.

The Inner Traffic Controller decides which virtual processor will run on the physical processor, based on the priority assigned to each virtual processor. Of course from the number of virtual processors assigned to a real processor only one can run on it at a time. The primitives ITC$AWAIT (Inner Traffic Controller AWAIT) and ITC$ADVANCE (Inner Traffic Controller ADVANCE) are used to provide communication and synchronization among the virtual processors. These primitives are very similar in form and function to the AWAIT and ADVANCE of the Traffic Controller.

III. MULTIPROCESSOR ARCHITECTURE

The manifestation of an operating system design is, of course, software in execution on a system of equipment. If the equipment must be selected early in the design, care must be taken to insure that the overall system design goals are compatible with the actual hardware capabilities. On the other hand, if specific design goals must be met, then actual hardware selection could be made late in the design process. Then, even if a hardware change must be made, the penalty for correcting it will be small, since only the lowest level of the design (where resources are virtualized), need be changed.

The particular hardware selected for this implementation, is based on the INTEL 86/12A single board microcomputer [2].

A. HARDWARE REQUIREMENTS

One of the principal design goals of the system design is to provide for configuration independence. That is when real processors are added or deleted the system will continue to function except of course for some change in performance. Therefore, the operating system imposes only a few constraints on the hardware, that are noted below:

1. Shared Global Memory

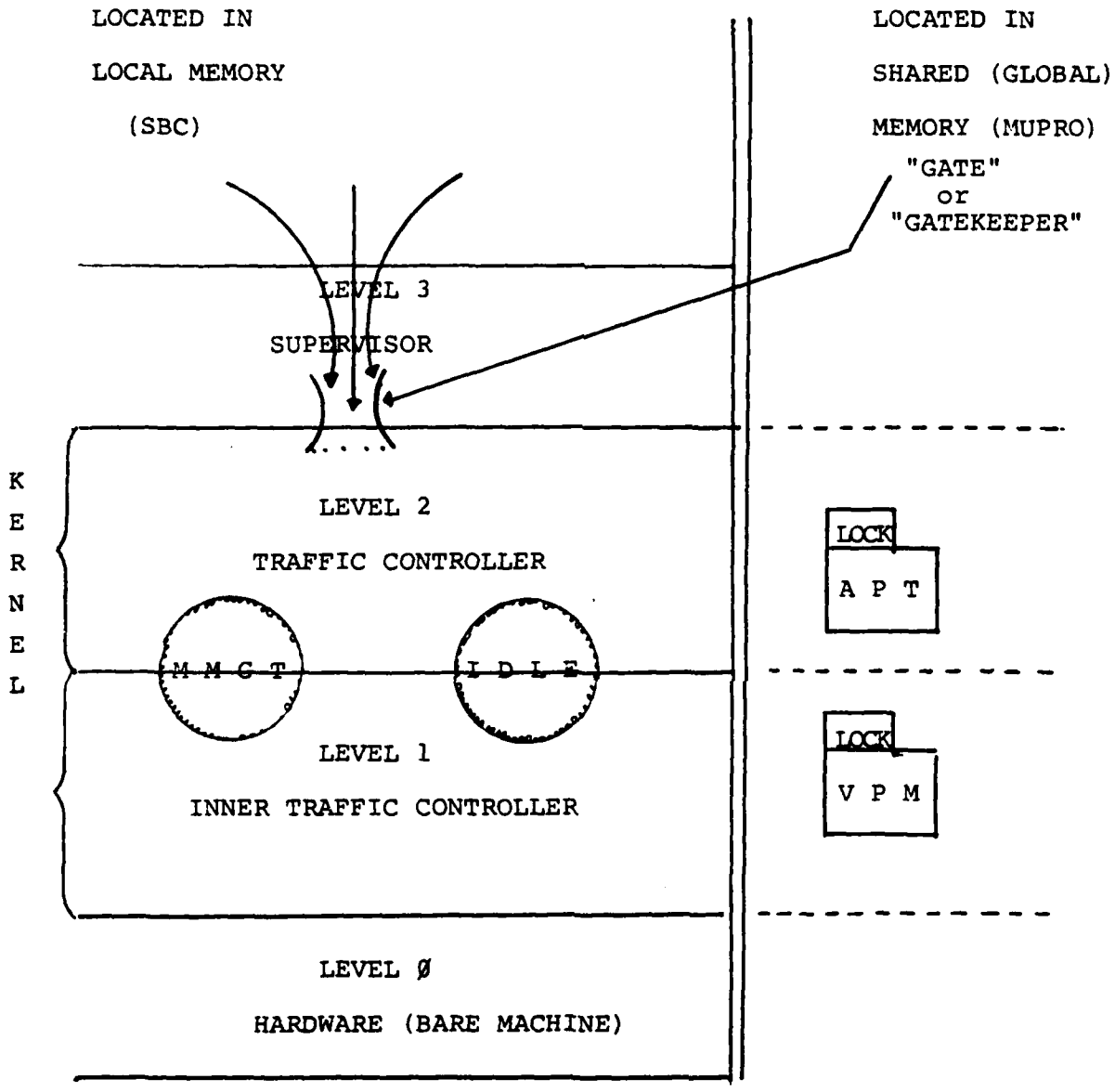
The operating system maintains, "system-wide control data" accessible to each of the processors via "shared"

segments. The communication path utilized for sharing this data is shared memory. Thus some shared memory must be made available to each microcomputer in such a way as to allow independent access at the level of single memory references. (a very small part, of a separate memory board (MUPRO) is used as shared global memory, in this system implementation).

2. Multiprocessor Synchronization Support

There must also exist some "hardware-supported multiprocessor synchronization primitive". This can be any form of an indivisible read-alter-rewrite memory reference. This capability is required, to implement the global locks on shared data to prevent race conditions, as the physical processors attempt to asynchronously manipulate shared data. For better understanding of this and the previous paragraph, consider the following cases of APT and VPM. Two of the system-wide data control tables are the VPM (Virtual Processor Mapping) and the APT (Active Process Table), as shown in Figure 9. VPM is the principal central data base for the Inner Traffic Controller which contains entries for all of the virtual processors in the system. Each entry (there is one "per virtual processor") has several fields, such as the virtual processor state, priority, etc., which will be described in Chapter IV.

Making this table globally available facilitates communication among virtual processors at the Inner Traffic



APT: ACTIVE PROCESS TABLE
VPM: VIRTUAL PROCESSOR MAP
SBC: SINGLE BOARD COMPUTER
MMGT: MEMORY MANAGEMENT PROCESS
IDLE: IDLE PROCESS

FIGURE 9. GLOBAL LOCKS ON SHARED CONTROL DATA

Controller level, on a "system-wide" scale, since every virtual processor can access this table.

But to prevent race conditions, and also to assure that only one processor, at a time, accesses the VPM, as the physical processors attempt to asynchronously manipulate shared data, we see the need of a global lock for VPM table. So in the implementation (coding), every time the VPM is accessed either to read data or to write, e.g. update a field, conceptually a "key" is turned and the VPM table is locked. When the access task is finished, before leaving we unlock the VPM.

Exactly the same concept is applied for the APT, which is the principal central data base for Traffic Controller in LEVEL 2, containing entries for every process.

In the implementation of this design (coding), we can see that modules accessing VPM, as for example, ITC$AWAIT and ITC$ADVANCE, and also modules accessing APT, as TC$AWAIT, TC$ADVANCE and TCPEHANDLER (Traffic Controller Preempt Handler), lock and afterwards unlock the corresponding global table.

To set these global locks, the implementation of the present design utilizes the "test-and-set semaphore" operation. This mechanism, supported by the PL/M built-in procedure "Lockset" [1], is a spin-lock with potentially significant impact on system bus traffic.

3. Inter-Processor Communication

Finally, some method of communication between physical processors must be provided. This is satisfied by an ability to generate interrupts between the physical processors. This capability is required for the implementation of "Preemptive scheduling" and is supported by the INTEL SBC 86/12A using a specific hardware configuration and software control.

B. HARDWARE CONFIGURATION

1. System Configuration

The hardware system is configured as a multi-processor [18]. It consists of a number of single board microcomputers and a global memory module, connected by a single shared bus. The system differs from conventional multiprocessors in that each of the microcomputers possesses its own local memory. The global memory module is connected directly to the system bus, and is the only physical shared memory resource by all of the processors. The general configuration is shown schematically in Figure 10.

2. Specific Hardware Employed

The particular hardware selected for this implementation is based on the INTEL 86/12A single board microcomputer [2]. This microcomputer utilizes the INTEL 8086 16-bit microprocessor capable of directly addressing a total of 1 Mega-byte of physical memory.

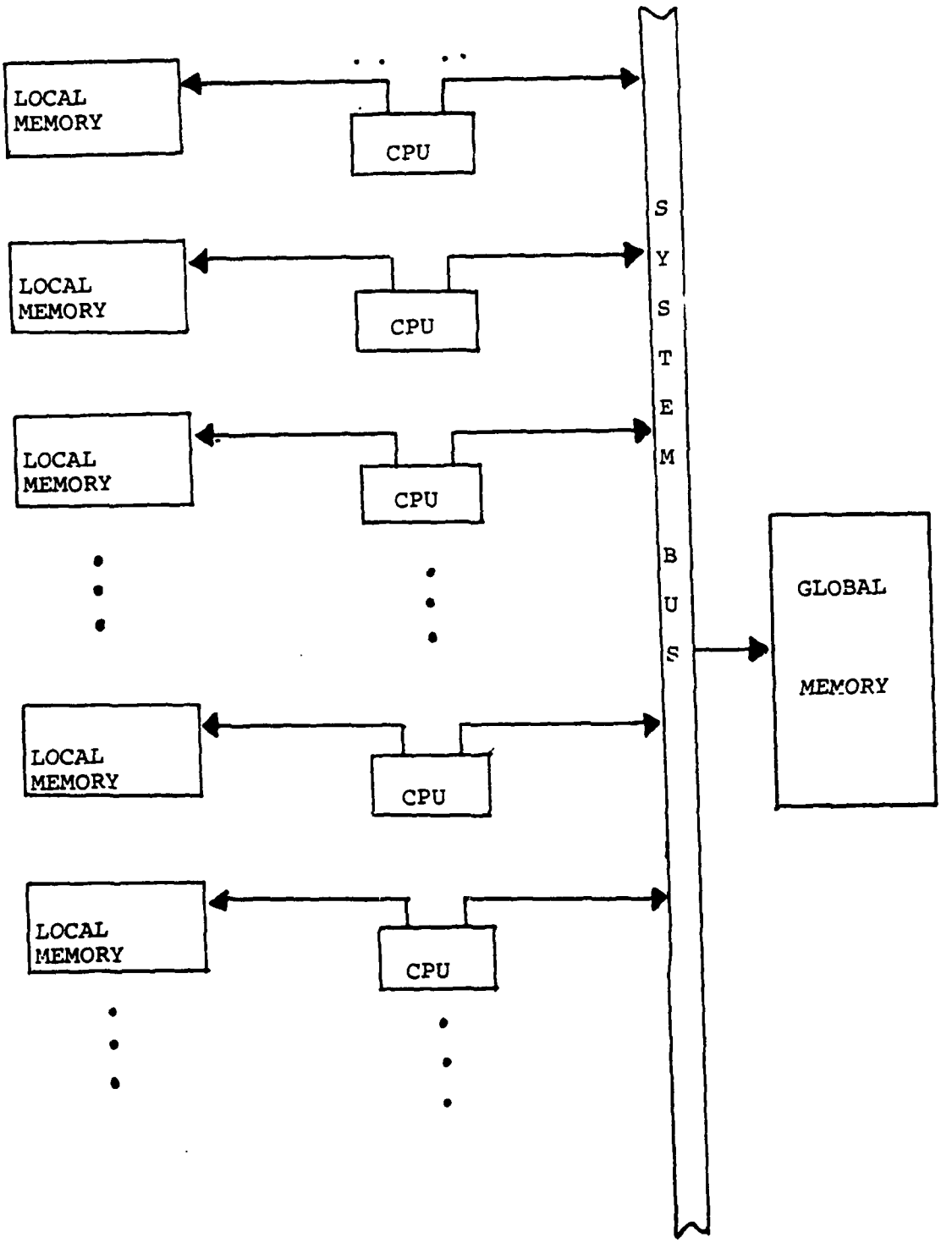


FIGURE 10. MULTIPROCESSOR CONFIGURATION

a. The 8086 Microprocessor

The 8086 microprocessor is suitable for a wide spectrum of microcomputer applications. Systems using 8086 can range from uniprocessor minimal-memory designs, to multi-processor systems with up to several Megabytes of memory.

The CPU is designed to operate with the 8089 input/output processor and other processors in multi-processing and distributed processing systems. Built-in coordinating signals and instructions, and also electrical compatibility with INTEL'S MULTIBUS shared bus architecture, support the development of multiple-processors design.

Actual performance, of course, varies from application to application. But in comparison to the 8-bit 2-MHZ 8080A, 8086 is seven to ten times more powerful. The high performance of the 8086 is realized by combining a 16-bit internal data path with a pipelined architecture that allows instructions to be prefetched during unused bus cycles. Furthermore software for high-performance 8086 systems need not be written in assembly language. The CPU is designed to provide direct hardware support for programs written in high-level languages, such as INTEL'S PL/M-86 which is used for the implementation of this operating system design.

The 8086 instruction set supports direct operation on memory operands, including operands on the stack. The hardware addressing modes provide straightforward

implementations of based variables, arrays, arrays of structures, character data manipulation (there is an extensive use of all these features in the implementation). Finally, routines with critical performance requirements that cannot be met with PL/M-86 may be written in ASM-86, the 8086 assembly language, and then linked with the PL/M-86 code. For example, the Virtual Processor Scheduler of the Inner Traffic Controller level is written in ASM-86.

b. Processor Architecture

Microprocessors generally execute a program by repeatedly cycling through the steps shown below:

(1) Fetch the next instruction from memory.

(2) Read an operand (if required by the instruction).

(3) Execute the instruction.

(4) Write the result (if required by the instruction).

The architecture of 8086, while performing the same steps, allocates them to two separate processing units within the CPU (see Figure 11). The execution unit (EU), executes instructions. The bus interface unit (BIU) fetches instructions, reads operands and writes results. These two units can operate independently of one another and are able, under most circumstances, to extensively overlap instruction fetches with execution.

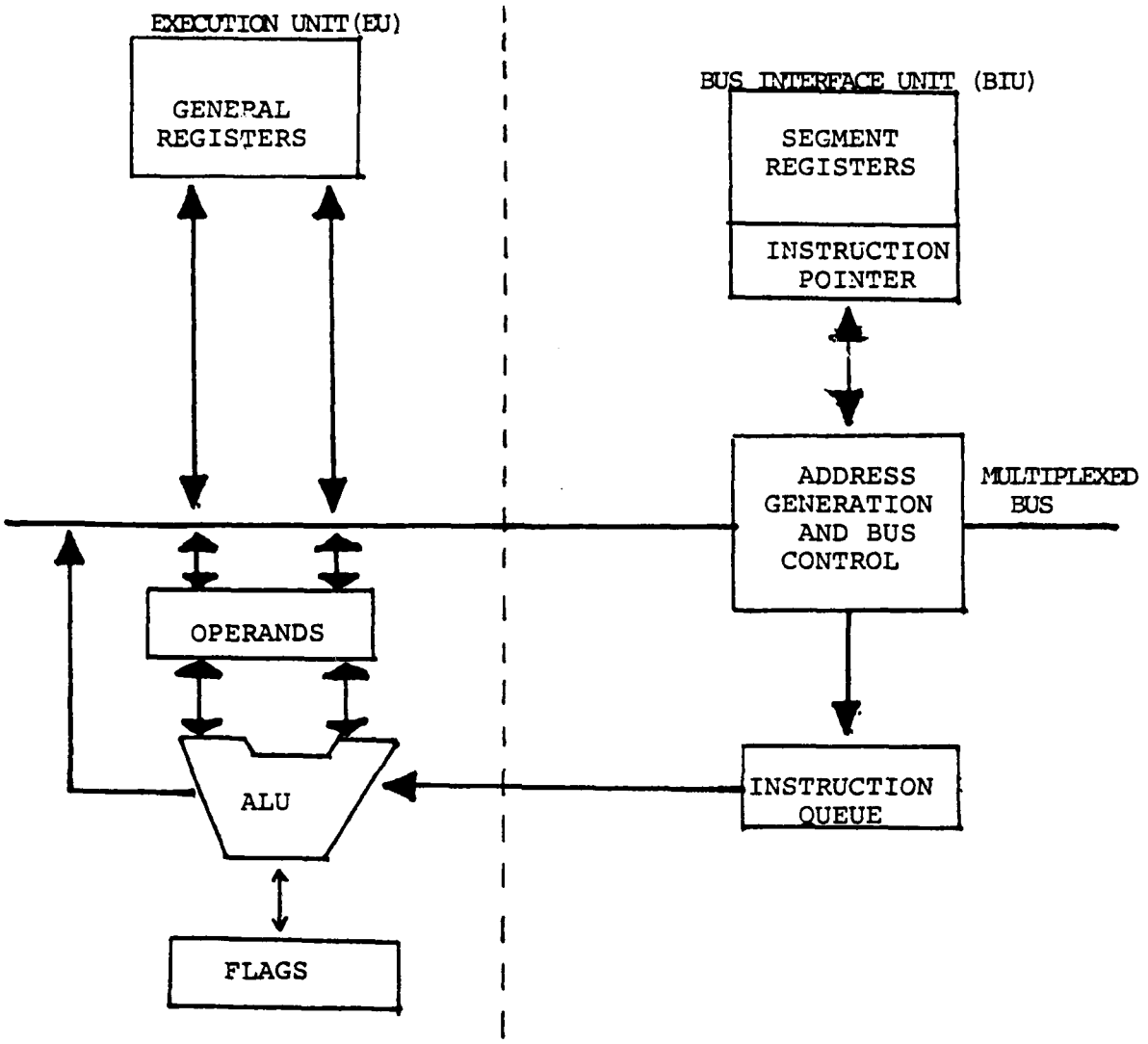


FIGURE 11. EXECUTION AND BUS INTERFACE UNITS (EU AND BIU)

The result is that, in most cases, the time normally required to fetch instructions "disappears", because the EU executes instructions that have already been fetched by the BIU.

A 16-bit arithmetic/logic unit (ALU) in the EU maintains the CPU status and control flags and manipulates the general registers and instruction operands. All registers and data paths in the EU are 16-bits wide for fast internal transfers. The EU has no connection to the system bus, the "outside world". It obtains instructions from a queue maintained by the BIU. Likewise when an instruction requires access to memory or to a peripheral device, the EU requests the BIU to obtain or store the data. All addresses manipulated by the EU are 16-bits wide.

The BIU performs an address relocation that gives the EU access to the full Megabyte of memory space. BIU performs all bus operations for the EU. Data is transferred between the CPU and memory or I/O devices upon demand from the EU. In addition, during periods when the EU is busy executing instructions, the BIU "looks ahead" and fetches more instruction from memory. The instructions are stored in an internal RAM array called the instruction stream queue (which can store up to six instruction bytes). This queue size allows the BIU to keep the EU supplied with prefetched instructions, under most conditions without monopolizing the

system bus. The BIU fetches another instruction byte, whenever there are two empty bytes in its queue and there is no active request for bus access from the EU (BIU normally obtains two instructions bytes per fetch).

Under most circumstances the queue contains at least one byte of the instruction stream, and so the EU does not have to wait for instructions to be fetched. The instructions in the queue are the next logical instructions as long as, execution proceeds serially. If the EU executes an instruction that transfers control to another location, then the BIU resets the queue and fetches the instruction from the new address, passes it immediately to the EU, and then begins refilling the queue from the new location. In addition, the BIU suspends instruction fetching whenever the EU requests a memory or I/O read or write (except that a fetch already in progress is completed before executing the EU's bus request).

c. CPU Registers

There are eight 16-bit general registers. The general registers are subdivided into two sets of four registers each. The first set, called the "H and L" group (for "high" and "low"), are the data registers. The second set, called the "P and I" group, are the pointer and index registers (see Figure 12).

The data registers have their upper (high) and lower (low) halves separately addressable. This means that

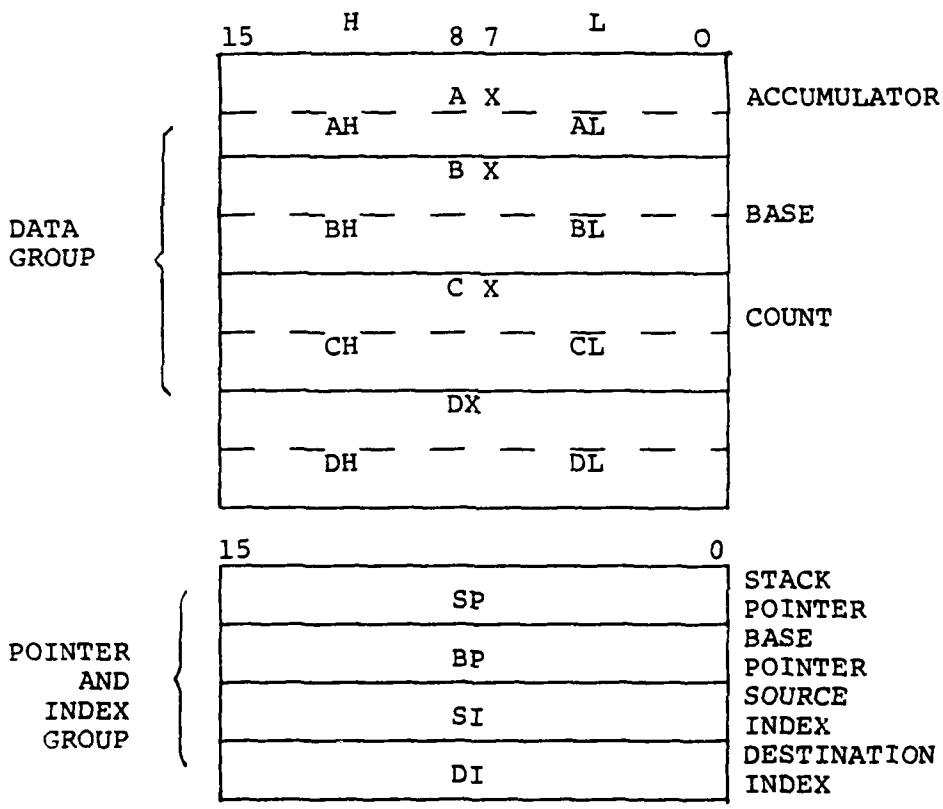


FIGURE 12. GENERAL REGISTERS

each data register can be used interchangeably as a 16-bit register or as two 8-bit registers. The other CPU registers always are accessed as 16-bit units only. The data registers can be used without constraint in most arithmetic and logic operations. In addition, some instructions use certain registers implicitly (see Figure 13), thus allowing compact yet powerful encoding.

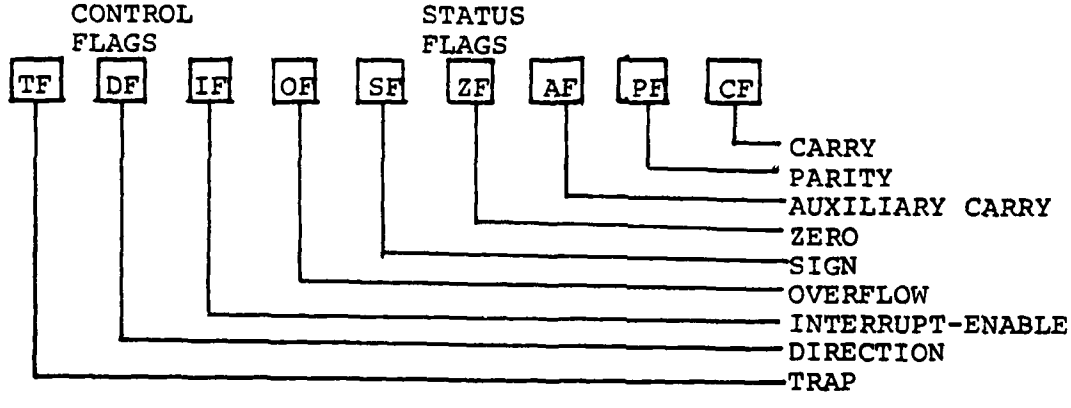
The pointer and index registers can also participate in most arithmetic and logic operations. The P and I registers (except for BP) also are used implicitly in some instructions, as shown in Figure 13.

The 16-bit instruction pointer (IP) (analogous to the program counter, PC, in the 8080 CPU) is updated by the BIU, so that it contains the offset (distance in bytes) of the next instruction from the beginning of the current code segment. IP points to the next instruction. During normal execution IP contains the offset of the next instruction to be "fetched by the BIU". Whenever IP is saved on the stack, it first is automatically adjusted to point to the next instruction to be "executed". Programs do not have direct access to the IP, but instructions cause it to change and to be saved on and restored from the stack.

The 8086 has six 1-bit "status flags" that the EU posts to reflect certain properties of the result of an arithmetic or logic operation. A group of instructions is

REGISTER	OPERATIONS
AX	WORD MULTIPLY, WORD DIVIDE, WORD I/O
AL	BYTE MULTIPLY, BYTE DIVIDE, BYTE I/O, TRANSLATE, DECIMAL ARITHMETIC
AH	BYTE MULTIPLY BYTE DIVIDE
BX	TRANSLATE
CX	STRING OPERATIONS, LOOPS
CL	VARIABLE SHIFT AND ROTATE
DX	WORD MULTIPLY, WORD DIVIDE, INDIRECT I/O
SP	STACK OPERATIONS
SI	STRING OPERATIONS
DI	STRING OPERATIONS

FIGURE 13. IMPLICIT USE OF GENERAL REGISTERS



IN GENERAL THE FLAGS REFLECT THE FOLLOWING CONDITIONS:

(1) AF: Auxiliary Carry Flag
(2) CF: Carry Flag
(3) OF: Overflow Flag
(4) SF: Sign Flag
(5) PF: Parity Flag
(6) ZF: Zero Flag
(7) DF: Direction Flag
(8) IF: Interrupt Flag
(9) TF: Trap Flag

FIGURE 14. FLAGS

available that allows a program to alter its execution depending on the state of these flags, that is, on the result of a prior operation. Three additional "control flags" can be set and cleared by programs to alter processor operations (see Figure 14).

d. Segmentation - Segment Registers

The 8086 does not support the notion of explicit segmentation. In the 8086, addressing is segmentlike, in that the base and offset (two-dimensional) addressing is used. 8086 programs "view" the one Megabyte of memory space as a group of segments that are defined by the application. A segment is a logical unit of memory that may be up to 64K bytes long. Each segment is made up of contiguous memory locations and is an independent separately-addressable unit.

Every segment is assigned (by software) a base address which is its starting location in the memory space. All segments begin on 16-byte memory boundaries. There are no other restrictions on segment locations. Segment may be adjacent, disjoint, partially overlapped or fully overlapped (see Figure 15). However, in this operating system design a physical memory location cannot be mapped on (contained in) more than one logical segment.

The segment registers point to (contain the base address values of) the four currently addressable segments (See Figure 17). Programs obtain access to code and data in

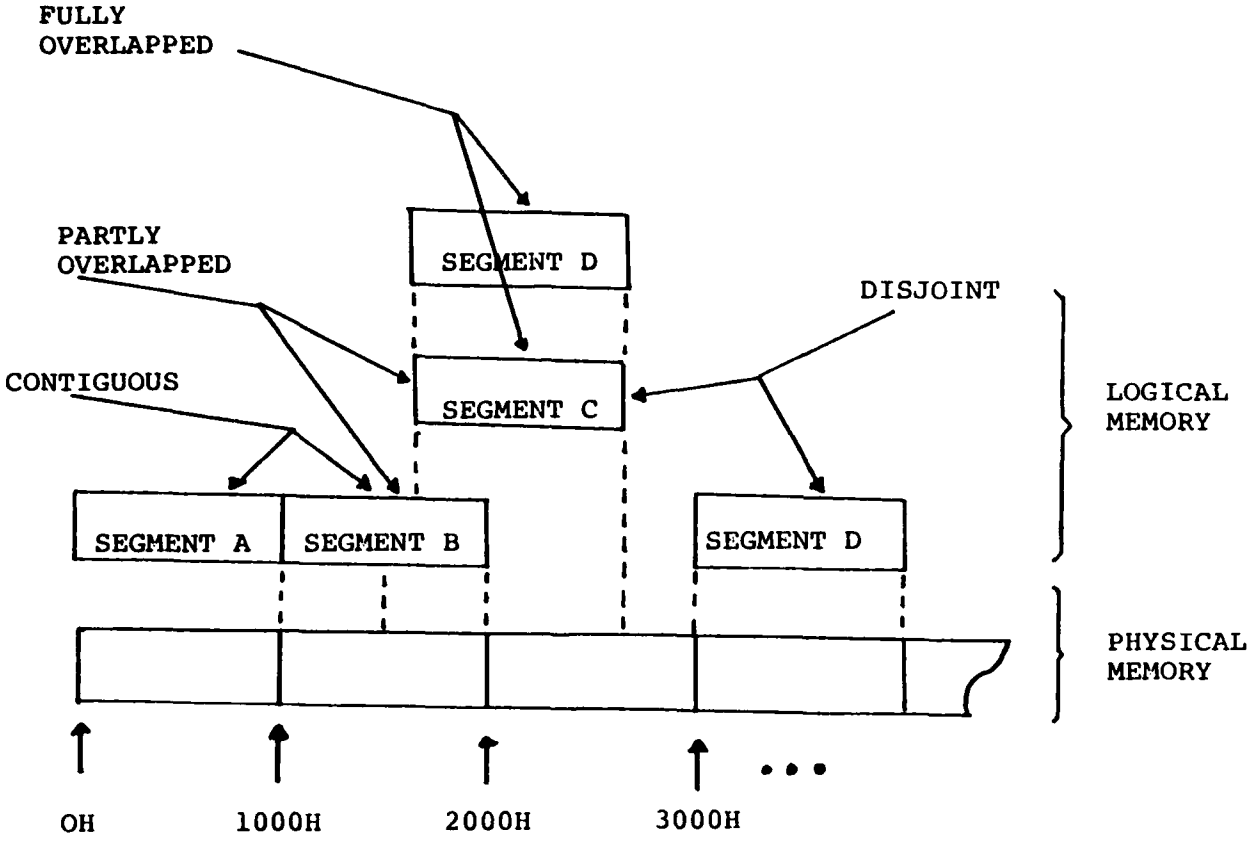


FIGURE 15. POSSIBLE SEGMENT LOCATIONS IN PHYSICAL MEMORY

other segments by changing the segment register, to point to the desired segment.

Every application can define and use segments differently. The currently addressable segments provide a generous work space of 64K bytes for code, 64K bytes for stack, and 128K bytes of data storage.

The CPU has direct access to four segments at a time. Their base addresses (starting locations) are contained in the segment registers (see Figure 16).

In this implementation these four base segment registers of the 8086 microprocessor are utilized as follows:

(1) Code Segment Register (CS register) is used for addressing a pure segment containing executable code. CS register points to the current code segment. Instructions are fetched from this segment.

(2) Data Segment Register (DS register) is used for processing local data. The DS register points to the current data segment that generally contains program variables.

(3) Stack Segment Register (SS register) is used for implementing the per process stacks (kernel stack and user stack). SS register points to the current stack segment. Stack operations are performed on locations in this segment.

(4) Extra Segment Register (ES register) is typically used for external or shared data. ES register

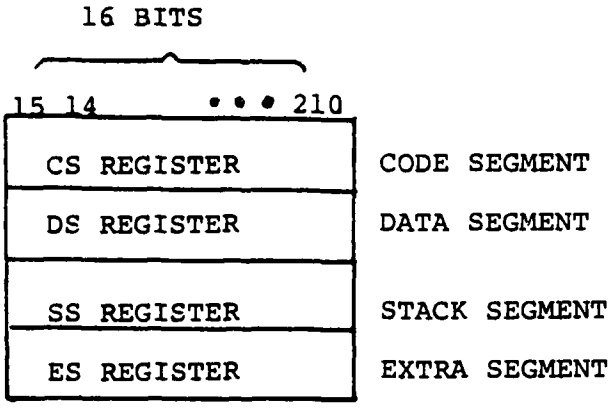


FIGURE 16. SEGMENT REGISTERS

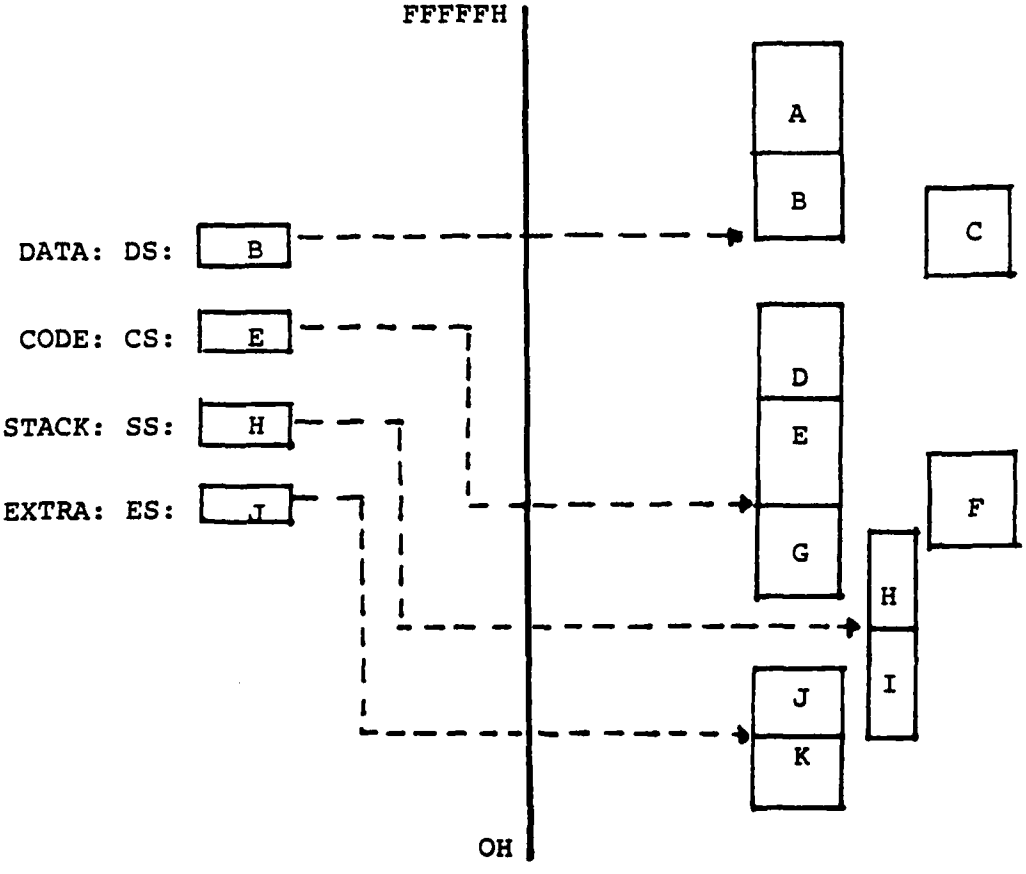


FIGURE 17. CURRENTLY ADDRESSABLE SEGMENTS

points to the current extra segment (which also is typically used for data storage).

In the 8086, a segment can range anywhere up to 64 kilo-bytes in length. Segments can be placed anywhere within the 1 mega-byte address space of the 8086 as long as the segment hexadecimal base is placed so that the last digit of the base is zero. Segment access and bounds checking are not supported. Although there is no general segmentation hardware, this design effects a segmented address space through a combination of operating system support and system initialization conventions described in a thesis by Anderson [19].

e. Physical Address Generation

It is useful to think of every memory location having two kinds of addresses, "physical" and "logical". A physical address is the 20-bit value that uniquely identifies each byte location in the Megabyte memory space. Physical addresses may range from OH through FFFFFH. All exchanges between the CPU and memory components use this physical address.

Programs deal with the logical rather than physical addresses and allow code to be developed without prior knowledge of where the code is to be located in memory this facilitates dynamic management of memory resources.

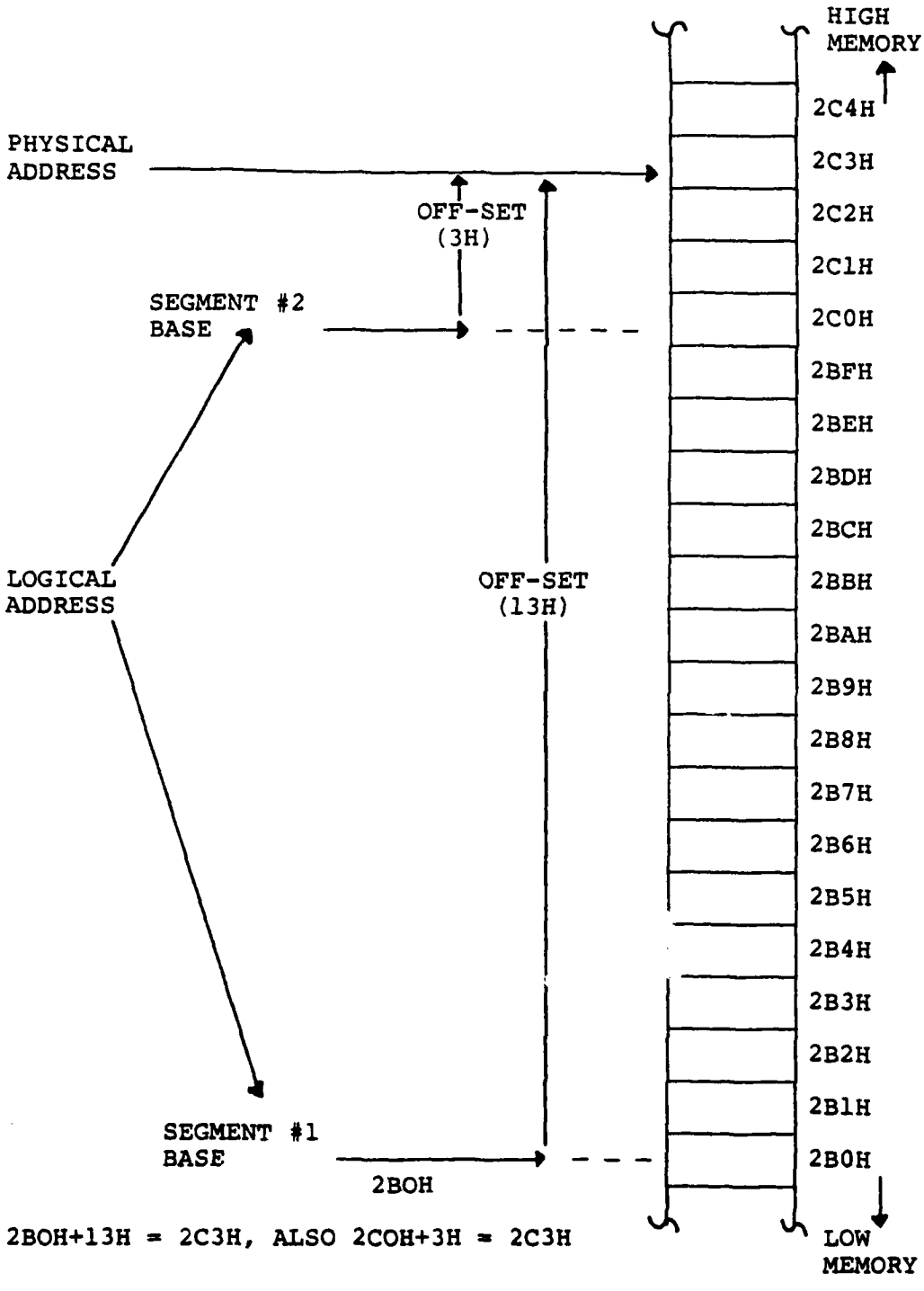


FIGURE 18. LOGICAL AND PHYSICAL ADDRESSES

A logical address consists of a segment base value and an offset value. For any given memory location the segment base value locates the first byte of the containing segment and the offset value is the distance in bytes of the target location from the beginning of the segment.

Segment base and offset values are unsigned 16-bit quantities. The lowest-addressed byte in a segment has an offset of 0.

Whenever the BIU accesses memory to fetch an instruction or to obtain or store a variable, it generates a physical address from the corresponding logical one. This is done (see Figure 19) by shifting the segment base value four bit positions to the left and adding the offset.

f. The iSBC 86/12A Single Board Microcomputer

The 86/12A is a complete computer capable of "stand-alone operation" used as the basic processing node of the multiprocessor. The iSBC 86/12A Board includes a 16-bit central processing unit (CPU), 32K bytes of dynamic RAM, a serial communications interface, three programmable parallel I/O ports, programmable timers, priority interrupt control, Multibus interface control logic, and bus expansion drivers for interface with other Multibus interface compatible expansion boards. Provision has been made for user installation of up to 16k bytes of read only memory (ROM). iSBC 86/12A is a

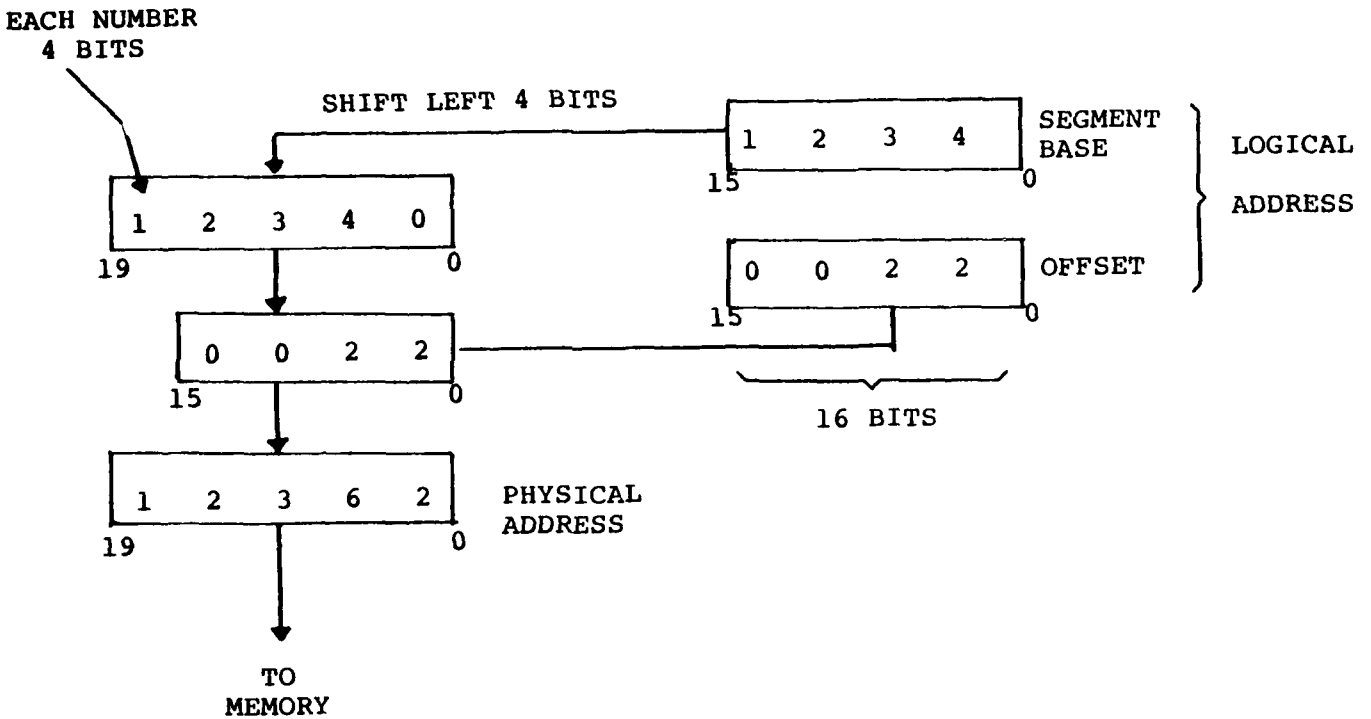


FIGURE 19. PHYSICAL ADDRESS GENERATION

commercial product which satisfies the three basic hardware requirements for this operating system mentioned in above subparagraph A (HARDWARE REQUIREMENTS). First, it must possess a system bus interface. Each microcomputer is capable of independently accessing a global shared memory via the system bus. Secondly, the 8086 CPU supports multiprocessor synchronization directly with an indivisible "test-and-set semaphore" instruction, which performs the bus lock. Lock semaphores reside in the shared global memory. Thirdly, preempt interrupts can be generated by using a bit of a parallel I/O port provided on each microcomputer. This requires connecting a bit of the microcomputer's parallel I/O port to the system interrupt structure.

g. Preempt Interrupt Hardware Connection

As with most microprocessors, the 8086 itself does not possess the capability to directly generate interrupts destined for other devices. The devices of interest here are the other processors. We need this capability for the implementation of preemptive scheduling. The system interrupt lines are accessible through a jumper matrix [2] located on the microcomputers. The parallel I/O output port of each iSBC 86/12A is connected to this interrupt jumper matrix. Preempt interrupts are then generated by the system simply by outputting a single word, through the parallel port, onto the system interrupt lines.

Note that only a single interrupt line is actually required to implement system-wide preempt interrupts. For details see the next chapter.

h. On Board Bus Structure - System Bus

The iSBC 86/12A board architecture is organized around a three-bus hierarchy: the on-board bus, the dual port bus, and the Multibus interface (see Figure 20). Each bus can communicate only within itself and an adjacent bus and also each bus can operate independently of each other. The on-board bus connects the CPU to all on-board I/O devices, ROM/EPROM, and the dual port RAM bus. Activity on this bus does not require control of the outer buses, thus permitting independent execution of on-board activities. Activities at this level require no bus overhead and operate at maximum board performance.

The next bus in the hierarchy is the dual port bus. This bus controls the dynamic RAM and communicates with the on-board bus and the Multibus interface.

When the on-board bus needs the Multibus interface, it must go through the dual port bus to the Multibus interface. The iSBC 86/12A Board is completely Multibus interface compatible and supports both 8-bit and 16-bit operations.

The Intel MULTIBUS [2] is utilized as the system bus. It is a widely used commercial product with a published

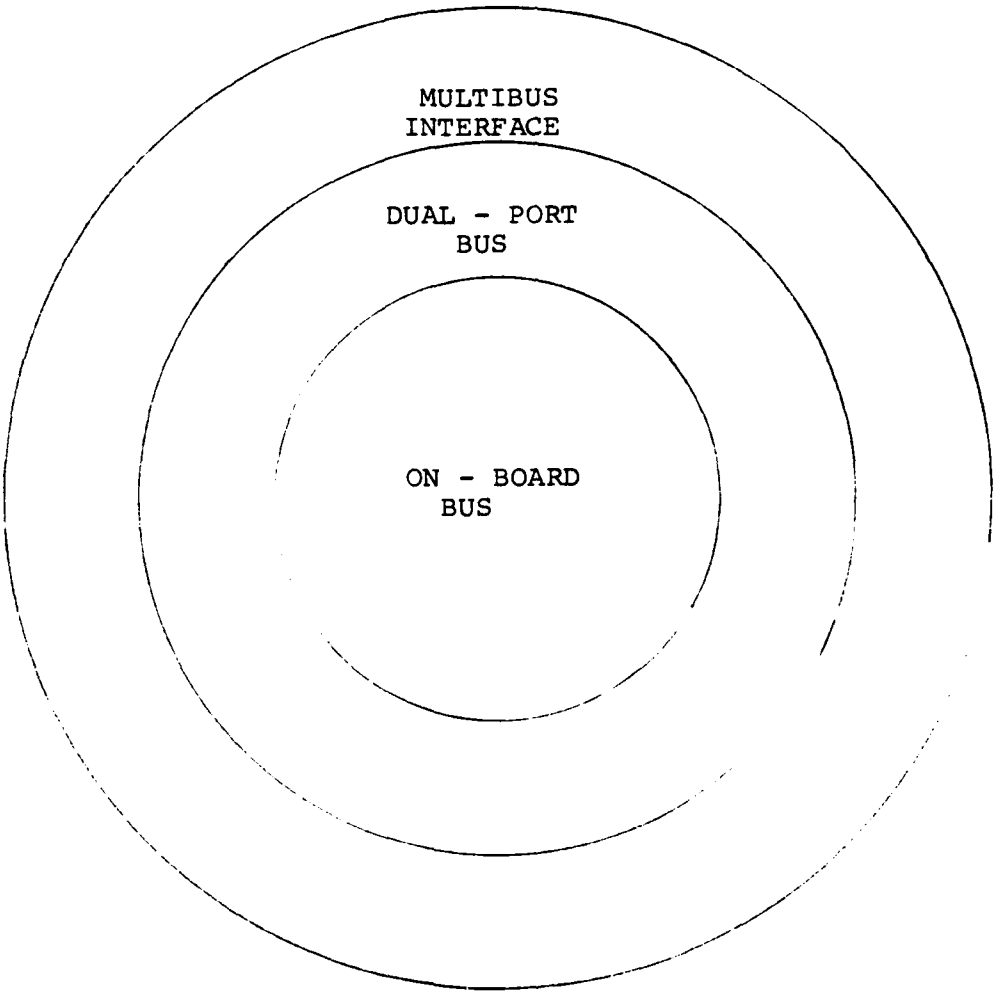


FIGURE 20. INTERNAL BUS STRUCTURE

set of standards. This bus is specifically designed to support multiple processors and is fully compatible with the microcomputers used. It is utilized without modifications.

C. HARDWARE ASSESSMENT

The commercially available 86/12A single board microcomputer was chosen because it was specifically designed to provide support for multiple processor systems. In using the operating system described in the next chapter to manage the microcomputer's physical resources, this microcomputer is entirely suitable for use as a basic processing node of an effective multiprocessor system. For multiprocessor interconnections see Figure 21.

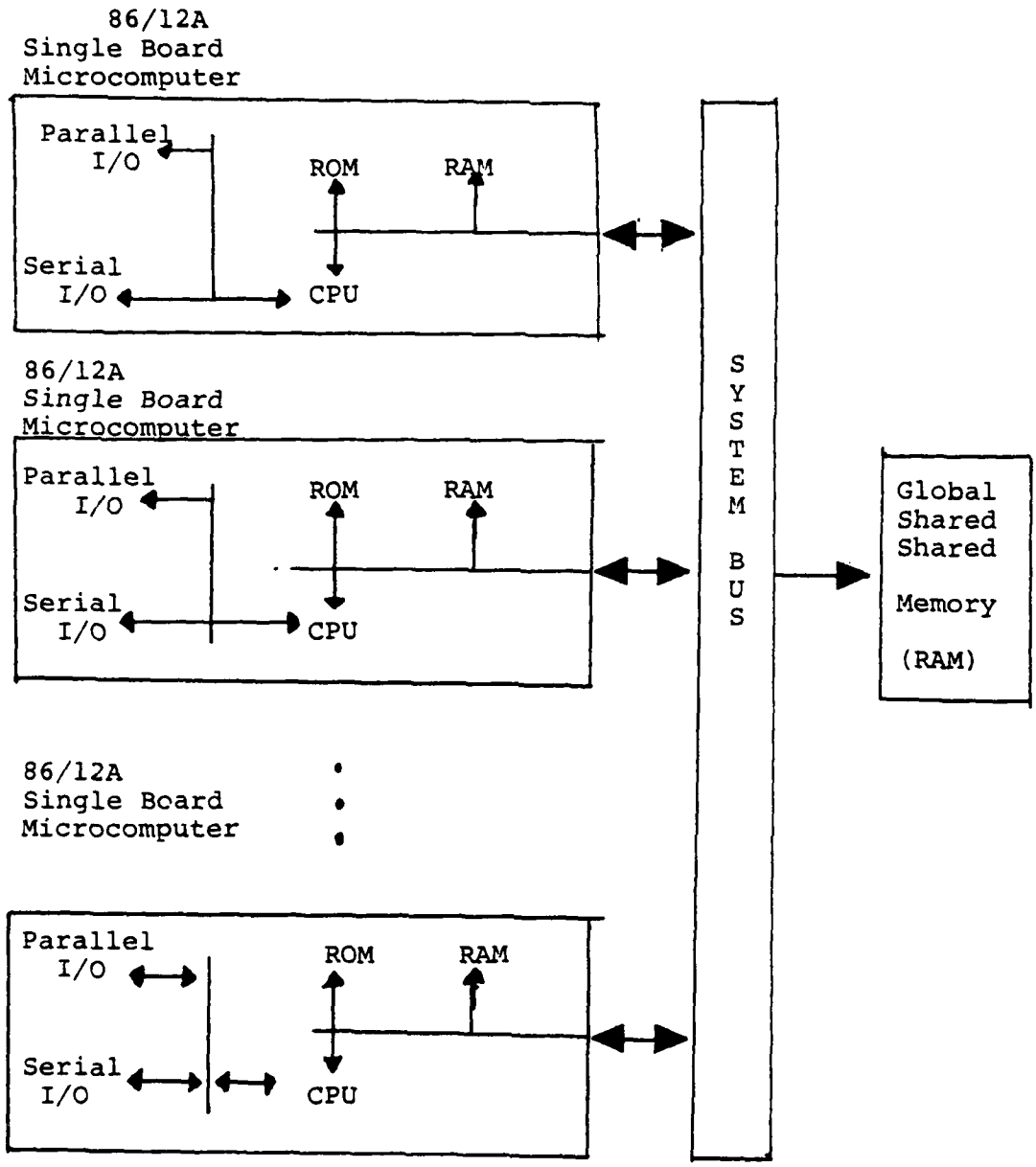


FIGURE 21. MULTIPROCESSOR CONFIGURATION

IV. DETAILED SYSTEM DESIGN AND IMPLEMENTATION

A. STRUCTURE OF THE OPERATING SYSTEM

The distributed modules of the kernel create a virtual machine hierarchy which controls process interactions and manages physical processor resources. The kernel is not aware of the details of process tasks. It knows each process only by a name (as an entry in the Active Process Table) and provides processes with scheduling and inter-process communication and synchronization services based on this process identity.

The kernel is constructed in terms of layers of abstraction. Each layer, or level, builds upon the resources created at lower levels. The rules of abstraction described in Chapter II were applied to the design of this structure.

This operating system provides a multiprogrammed multi-processor system with segmented process address spaces using the hardware described in Chapter III. The operating system is structured as a hierarchy of four levels of abstraction, as follows:

Level 3: Supervisor

Level 2: Traffic Controller

Level 1: Inner Traffic Controller

Level 0: Hardware (Bare machine), (See Figure 5).

Level 0 is the bare machine which provides the physical resources (processors and storage) upon which the virtual

machine is constructed to give the extended machine view.

The remainder of this chapter will describe the level of virtualization (or layer of abstraction) created by each distributed kernel module.

The Inner Traffic Controller (Level 1) forms the first level of the hierarchy. It is "closest" to the hardware and encompasses the major machine-dependent aspects of the system. The Inner Traffic Controller multiplexes the physical processors among a pool of more numerous virtual processors.

Residing at the next level (Level 2) is the Traffic Controller, which is responsible for multiplexing the virtual processors among a larger number of user processes competing for resources. The user-accessible inter-process communication and synchronization primitives (Advance, Await and Ticket) provided at this level allow the user to easily satisfy complex system-wide inter-process synchronization requirements.

The Supervisor resides at the topmost level (Level 3). The Supervisor's purpose is to provide common services for user processes. In this implementation it only provides a simple assembly language interface to the kernel by having a single entry point into the kernel (the Gate or Gatekeeper).

B. CONTROL OF PROCESSOR MULTIPLEXING

There are two common schemes for the control of processor multiplexing: "centralized control" and "distributed control".

Centralized control is based on the idea of a central agent which is responsible for the binding of virtual processors to real processors. All these bindings are caused by the action of the central agent. This agent can be viewed as a process, since it is a sequential computation that performs operations on the state of the system. In this scheme of control usually this central agent is permanently bound to a dedicated real processor. Of course this implementation requires some kind of communication channel between the real processors and the central agent.

The main advantage of the centralized algorithm is, "unity". Since the centralized scheme is executed as a process permanently bound to one real processor, it can be described by a single sequential program that makes one decision at a time (that means a simply structured processor multiplexing policy).

An alternative scheme for the control of processor multiplexing is one in which the functions are accomplished by a distributed algorithm, executed by each process on all real processors.

The main advantage of the distributed scheme is, "autonomy". This autonomy afforded by a distributed system can increase the amount of parallel activity (real processors can execute in parallel). This scheme also results in a uniform design that is identical for every processor.

The advantages of each scheme are disadvantages of the other. In the centralized case the lack of autonomy prohibits

the parallelism afforded by the distributed scheme. On the other hand, in the distributed case, the autonomy makes it potentially difficult to understand the interaction of the multiplexing algorithm executed by different real processors.

1. Distributing the Operating System

One of the primary concerns in any multiple computer system is the issue of performance. The type of system in the present implementation is a multiprocessor with a "single" shared system bus. Thus the most glaring potential "bottle-neck" is the system bus. Thus it becomes desirable to minimize accesses to this resource which must be shared by all of the real processors.

The decision was made to "distribute" the operating system logically and physically to reduce the "system bus" use. Logically the segments of the operating system kernel are distributed within (as part) the address space of each user process. On the other hand, the performance issue is dealt with by physically distributing copies of the kernel in the local memories of each of the real processors. This allows high-speed access to kernel functions without necessitating the heavy use of the system bus for code fetches thus reducing "BUS contention".

Since the operating system is small, the memory wasted by distributing a copy of the kernel to each single board computer is a small price to be paid to allow performance to grow with the addition of real processors.

Thus, each computing node can be regarded as "semi-autonomous" in that each of the processors schedules itself. The modes are still centrally controlled by the set of system-wide data tables, kept in the global memory, which provides access to all real processors and thus eliminates the need of a central controlling process. The amount of memory needed for these system-wide data tables is almost negligible.

In this implementation there is no notion of a master-slave relationship among individual microcomputers, nor are individual kernel functions divided among them, as is commonly done. Rather, the "entire" kernel is distributed on each single board computer.

C. REAL TIME PROCESSING

Real-time control systems are designed for handling data within a time period which is consistent with the response time demanded by the process which generated the information. Such systems operate in a multi-programmed environment where the execution of a number of tasks is determined by the software priorities, hardware interrupts, timing algorithms and requests from other tasks (requests from one task to start, suspend or terminate another task) to pass data from one task to another.

A real-time operating system must be designed so that it is impossible for any program and any user to interfere with the execution of critical tasks by halting the machine, by

changing interrupt priorities or by innappropriately overwriting memory.

Real-time processing involves the performance of time-critical processing often related to the control of external devices. This application requires that some mechanism be employed to ensure that the time-critical processing is given immediate attention.

The hardware-supported "process preemption" mechanism employed in the system provides the rapid response required for real-time processing. The priority-driven preemptive scheduling technique used provides for expeditious handling of processes which perform the time-critical functions. These processes are assigned high priorities so that the system will preempt other processes of lower priority which may be in the running state. Thus when one of these high-priority processes is signalled, it can be immediately scheduled and thus gain control of the processor resources.

The actual system response time for a task request depends mainly on whether or not another task is running at a higher-priority level. To prevent high-priority tasks from executing too long, a "watchdog timer" is often used to guarantee that all tasks are serviced. This timer is set at the start of each task with the maximum duration that a task may run at a particular priority level before being suspended or dropped. This watchdog timer is not yet implemented but it will be a useful added capability.

D. SCHEDULING

Processor multiplexing and process multiplexing require a policy, called the Processor/Process multiplexing policy algorithm, or simply "Scheduling" algorithm.

In this design the scheduling functions are divided between the Inner Traffic Controller and the Traffic Controller levels. The Inner Traffic Controller multiplexes Virtual Processors among Real Processors. Each Real Processor possesses a fixed number of Virtual Processors (4 in the current version). At any one time the Real Processor can only execute "one" Virtual Processor. The choice of the Virtual Processor that will run in each Real Processor is decided by the Virtual Processor Scheduler (VPSCHEDULER) that is a routine in the Inner Traffic Controller level.

The Traffic Controller multiplexes Processes among Virtual Processors. The Traffic Controller Scheduler (TC$SCHEDULER) is responsible for that scheduling. Both scheduling algorithms are "priority driven". (The highest priority Virtual Processor or Process will run first). These algorithms receive as input the set of Virtual Processors and Processes respectively, that can be run and choose the next one to run.

More details will be presented when we will discuss the algorithm of these two "scheduler" modules.

E. PROCESS ADDRESS SPACE

The address space of a process is a set of PL/M-86 segments such as procedures (code), local variables (data), external

data (shared data) and a stack. Physical memory is allocated to the segments of a process in such a way as to limit system bus contention, as discussed by Anderson [19]. In this implementation the concept of a "per process stack" is a key element in the management of processes.

1. The PL/M-86 Stack

Intel's high level language PL/M-86 [1,6] utilizes the stack segments to implement per process stacks. Addressing of stacks is accomplished by using three of the 8086's registers as shown in Figure 22. The Stack Segment (SS) Register contains the base location of the stack segment in memory. The Stack Pointer (SP) Register addresses the current top of the stack as an offset from the base of the stack segment, (the value in the SS Register). The Base Pointer (BP) Register also holds an offset from the SS Register and is used to establish the procedure activation records [3, 4, 5]. During the "process creation" in the current version of the operating system one of the parameters passed to the operating system for a specific process is the initial value of the SP register ("maximum stack length"). It is used to assure no "stack overflow". If the process has only one module, the "maximum stack length" can be extracted for the specific process during its preparation (Compilation-Linking-Locating). Specifically the LST output file of the Compiler (file name.LST) at the end provides the information illustrated in Figure 23.

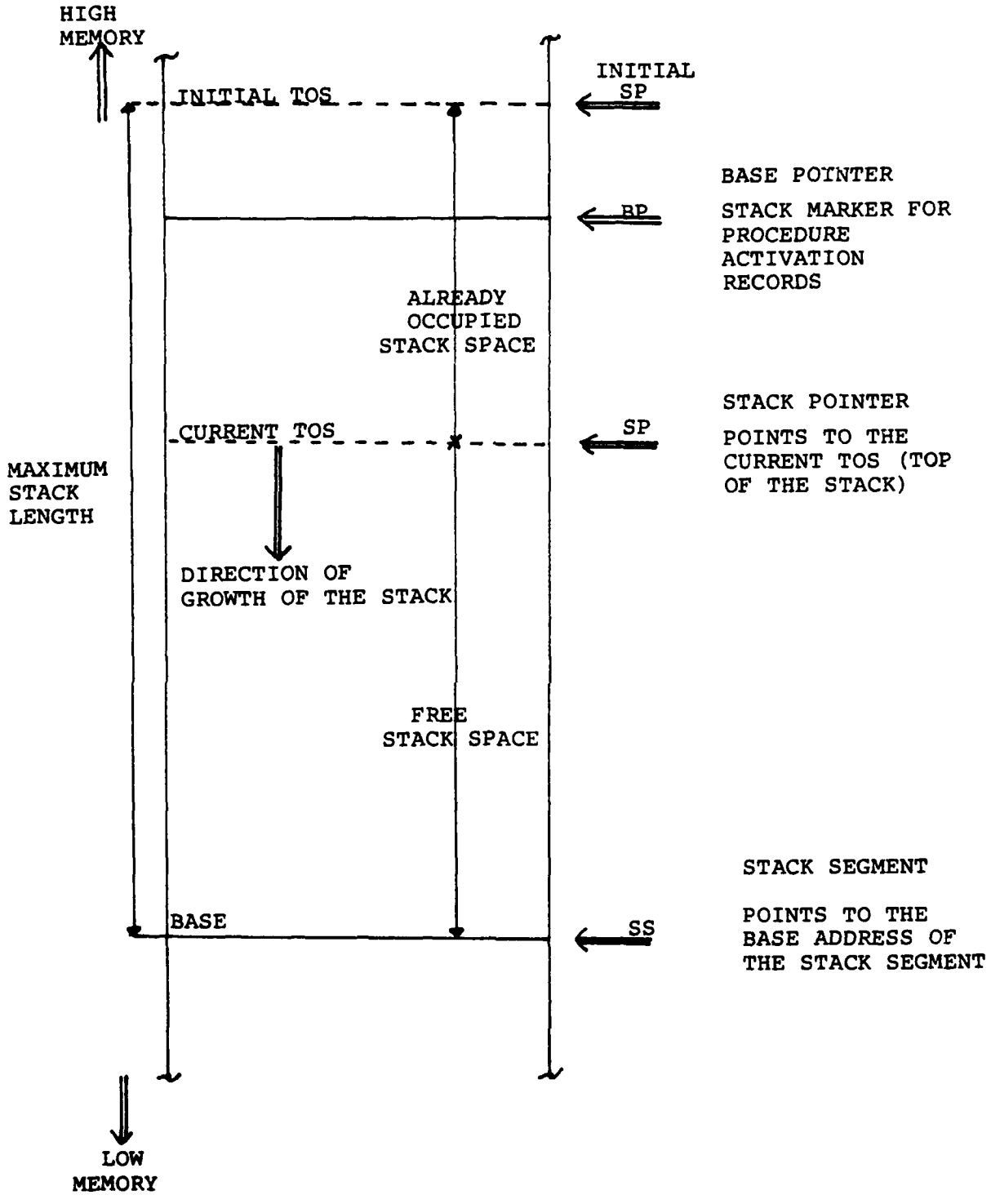


FIGURE 22. PL/M STACK STRUCTURE

There is also a second way to extract this information using the MP2 output file of the Locater (file name.MP2). This method can be used also when a process consists of

```
MODULE INFORMATION:

     CODE AREA SIZE      = 013BCH    4540
     CONSTANT AREA SIZE  = 0000H       0
     VARIABLE AREA SIZE  = 0276H     1142
     MAXIMUM STACK SIZE  = 0024H       36
     137 LINES READ
     2 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

FIGURE 23. MAXIMUM STACK SIZE (SAMPLE LST COMPILER OUTPUT)

several modules linked together. At the end of MP2 file is found the maximum stack size as shown in Figure 24.

```
MEMORY MAP OF MODULE INITINT
READ FROM FILE :F1:TESTIN.LNK
WRITTEN TO FILE :F1:TESTIN

MODULE START ADDRESS  PARAGRAPH = 2020H  OFFSET = 0004H
SEGMENT MAP

START     STOP      LENGTH ALIGN NAME              CLASS

002000H   0035BH     013BCH   W   INITINT_CODE     CODE
00276CH   0276FH     0004H    W   INITINT_DATA     DATA
007700H   007923H    0024H    W   STACK            STACK
007940H   007940H    0000H    W   MEMORY           MEMORY
```

FIGURE 24. MAXIMUM STACK SIZE (SAMPLE MP2 LOCATER OUTPUT)

To obtain the above "maximum stack size" information the command "COPY :F1: File name.LST TO :CO:" is typed on the MDS

(INTEL'S Microcomputer Development System) after the compilation or the command "COPY :F1: File name.MP2 TO :CO:" after the Locating Process. These commands will present the Compiler or Locater output file on the CRT screen of the MDS. If one prefers to have this output on the printer, just change ":CO:" to ":TO:".

In the same way we can extract the information on the maximum stack size of the kernel.

In the current version of the operating system, we use two "per process" stacks dividing the "address space" of each process into two "domains" of execution and separating the "user domain" from the "kernel domain". We call the corresponding stacks the "user stack" and the "kernel stack". In this version:

Maximum Kernel Stack Length =

Maximum Stack Size for the Kernel +10 and

Maximum User Stack Length =

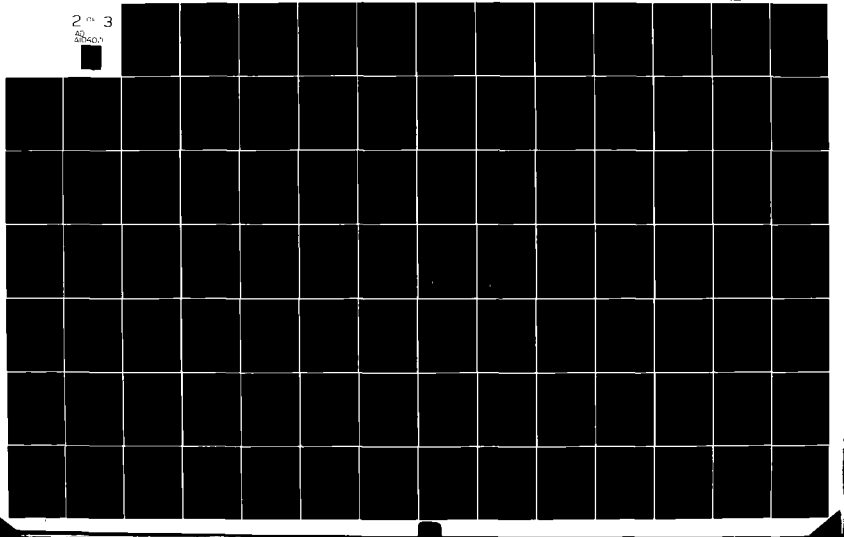
Maximum Stack Size for the "User Program""Linked" with the "Gate" + 10.

This value 10 is used to avoid overwriting the "stack header" shown on Figure 26 which occupies the first words in the stack, just above the stack base. It is important to make a distinction between the "User Program" or "Application Program" and the "User Process" or Application Process". The User or Application program is the "job" submitted to the operating system and the User or Application Process is this

AD-A104 071 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
DETAILED DESIGN AND IMPLEMENTATION OF THE KERNEL OF A REAL-TIME--ETC(U)
MAR 81 D K HAPANTZIKOS

UNCLASSIFIED NL

2 OF 3
AD
A104071



program in execution on a processor and is the submitted job linked with the distributed part of the Operating system. In Figure 25, we can see the User Program as the "subset A" and the distributed part of the operating system as the "subset B".

The running User Process (this program in execution) is the "Union C" of these subsets A and B, (C = AUB). This connection (linking) of the job with the operating system is accomplished via the "Gate."

2. The Stack as the "Address Space Descriptor"

In this system the per process stacks are used to maintain the process state information. This includes the current execution point (when the process is not actually running) and the locations of the code and data segments. This allows the system to "swap" in a new address space (viz., do a "context switch") by changing "only" the value in the SS Register which is thus used in a manner somewhat analogous to the MULTICS "Descriptor Base Register" (DBR) [9]. Then the operating system finds the remaining of the needed information to run the specific Process inside its stack.

Figure 26 shows how this information is stored in the kernel stack while a process is not actually running on a physical processor. The Base Pointer and Stack Pointer are stored in reserved locations at the very beginning of the stack segment, ("header" of the Stack). Figure 26 illustrates the status of the kernel stack after an interrupt within the kernel.

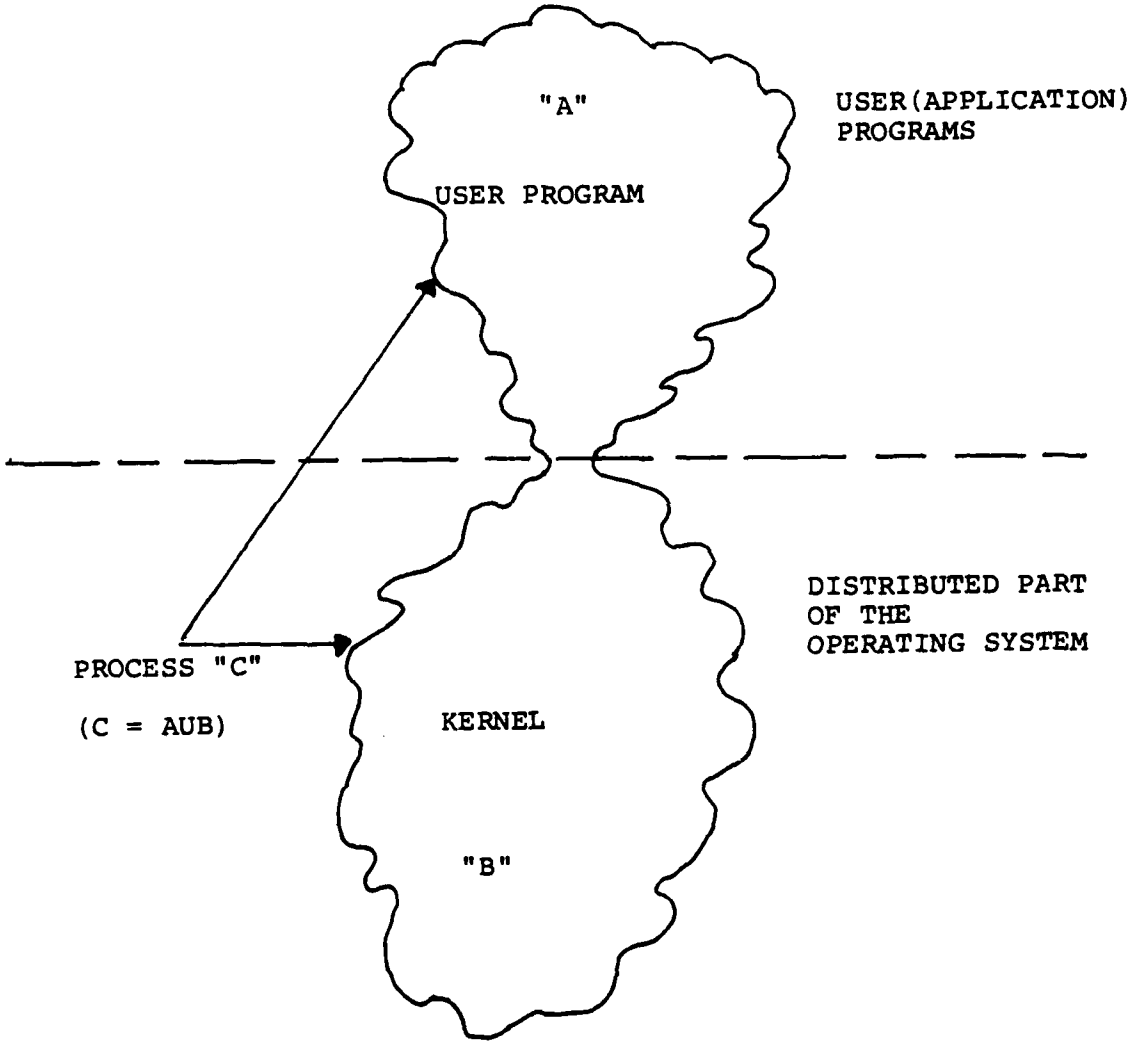


FIGURE 25. DISTINCTION BETWEEN A PROGRAM AND A PROCESS

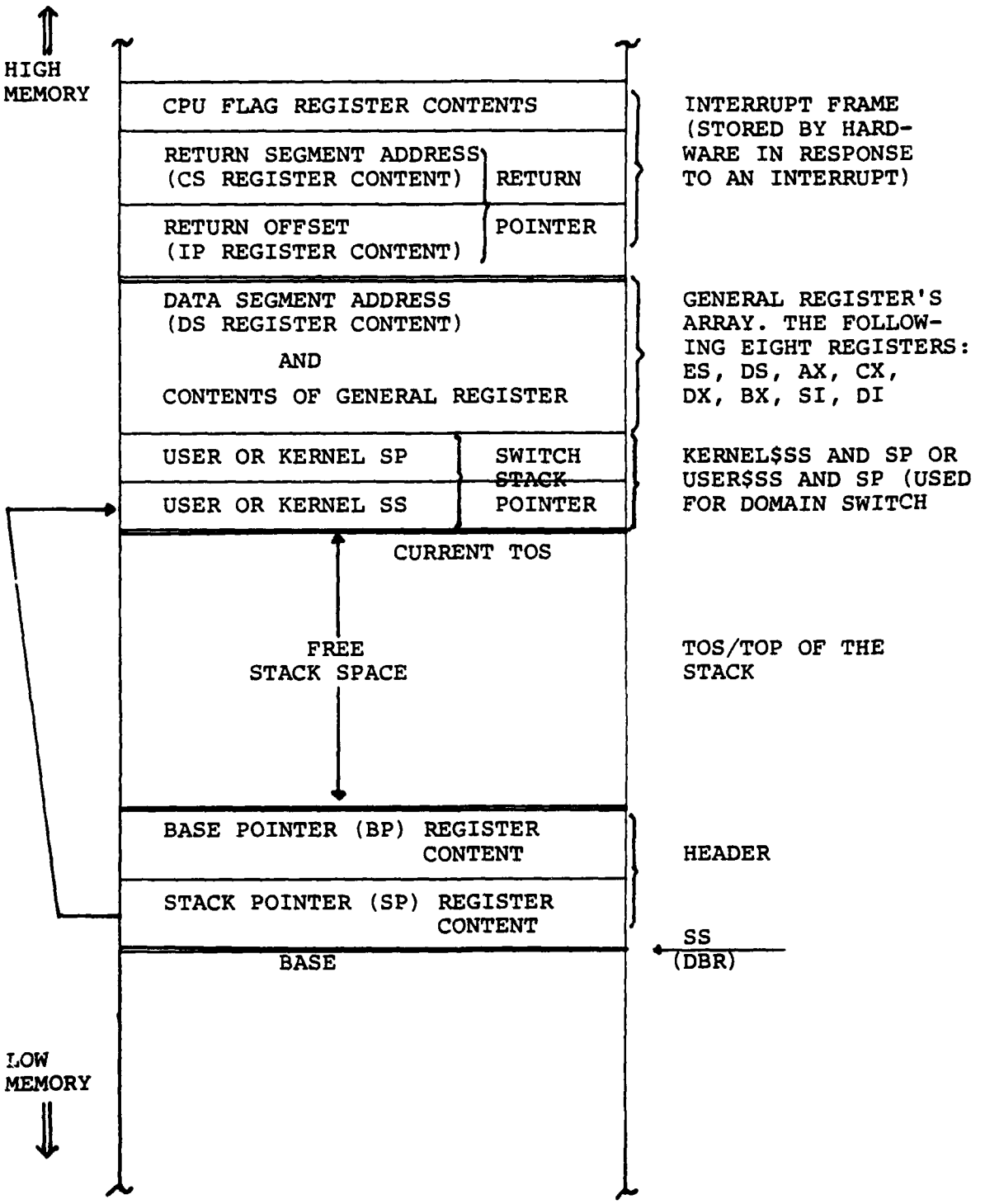


FIGURE 26. THE KERNEL STACK AS THE "ADDRESS SPACE DESCRIPTOR" (AFTER RESPONDING TO AN INTERRUPT)

In order to identify the stack segment and thus access the address space of a process, the stack segment base address is used in a dual role. First, a "unique base address" is assigned to the stack of each process which provides a "unique segment" for each stack. This base address is used for addressing locations within the stack. Secondly, the base address serves as a descriptor for the address space of each process. Thus the binding of a processor is changed from one process to another "merely" by changing the base address, viz., changing the value in the Stack Segment (SS) Register. Figure 26 illustrates how the "per process" Kernel Stack is implemented in the current version of the Operating System. More details about the currently used Stack mechanism (two per process stacks, two domains of execution) will be discussed when we describe the "Create Process" module of the Traffic Controller.

F. COMMUNICATION AND SYNCHRONIZATION

1. Process Synchronization

The problem of process synchronization arises from the need to share resources in a computer system. This sharing requires coordination and cooperation to ensure correct operation. This coordination is forced upon the processes by the operating system because of the scarcity of resources, for example, the need to wait for access to an I/O channel. In other cases a simple job may consist of several interactive processes, such as an airline reservation system.

Associated with processor allocation and interprocess synchronization are two synchronization problems, "race condition" and "deadly embrace" or "deadlock situations".

2. "Race Condition"

A race condition occurs when a desired action cannot be completed in one indivisible step. For example, in order to gain exclusive control of a printer in Process 1 in Figure 27, it is important to check if the printer is already in execlusive use by Process 2. If a flag is used (F=0, not in use)(F=1, in use) to indicate whether or not the printer

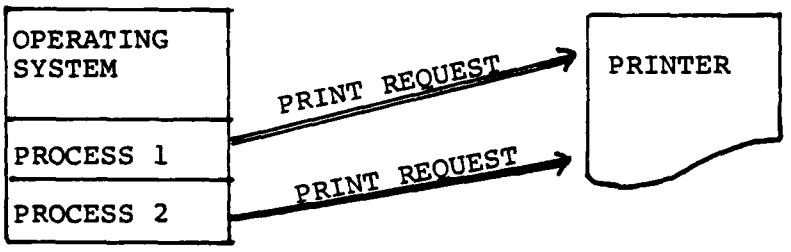


FIGURE 27. A SIMPLE RACE CONDITION

is in execlusive use, then this flag has to be interrogated. If Process 1 interrogates F and finds its value is zero, then it can set the value of F to one and enjoy the execlusive use of the printer. A problem arises when both Process 1 and Process 2 nearly simultaneously interrogate the flag in the following sequence:

Process 1 F=0? Yes

Process 2 F=0? Yes

Process 1 Sets F=1

Process 2 Sets F=1.

In this case both processes falsely gain the impression that they have exclusive use of the printer. This so called "race condition" can be avoided by an indivisible test and set operation which would prevent Process 2 being mislead. Such a test and lock operation is the 8086 LOCKSET built-in procedure.

In addition to physical devices, there are other shared resources, such as a shared database, that require the same type of synchronization to avoid race conditions. For example, in this implementation in order to avoid race conditions in the shared databases APT and VPM, we implemented a lock per database. When a Virtual Processor needs to read or update the shared database, it locks this common table (this way it locks out all the other Virtual Processors). After the completion of this action the Virtual Processor unlocks the database, so another Virtual Processor can access it.

3. "Deadly Embrace" or "Deadlock Situations"

A "deadly embrace" is a situation in which two processes are unknowingly waiting for resources that are held by each other and thus unavailable [15]. See Figure 28.

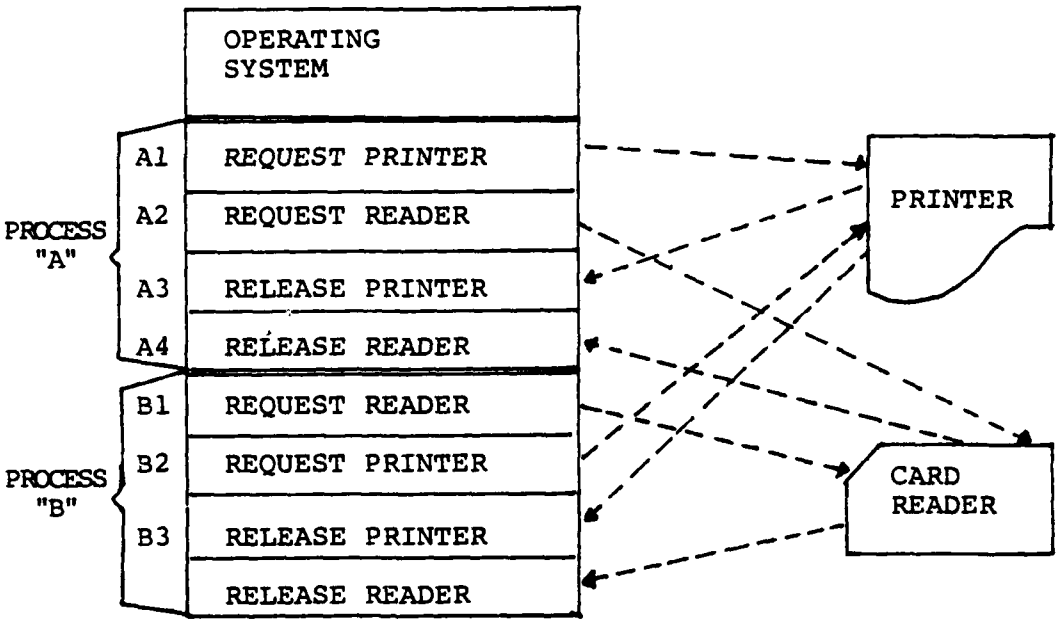


FIGURE 28. DEADLY EMBRACE SITUATION

"A" and "B" are sharing the use of the printer and card reader by means of the request and release operations (as stated in the previous paragraph). Due to independent scheduling of the processes the "request" and "release" operations may be interspersed in several different orders.

Lets consider a case that starts with A1 (request printer for process "A") and B1 (request reader for process "B"). If then A2 occurs (request reader for process "A"), process "A" must be blocked because the reader is already in use by process "B". Then when B2 occurs (request printer

for process "B"), process "B" must also be blocked because the printer is already in use by process "A". In this way we confront a situation where each process is waiting for the other to release a needed resource. A deadly embrace situation is resulted.

We have already concluded that synchronization primitives like the described "request" and "release" cannot avoid "race conditions" and "deadlock situations". Several more sophisticated synchronization primitives have been developed to overcome these problems. The most commonly used among them are: Dijkstra's "P" and "V" operations on "counting semaphores" [12], Saltzer's "Block-Wakeup" or sometimes called "Wait-Signal" [14] and lastly Kanodia and Reed's "Eventcounts" and "Sequencers" [10].

4. Shared Segment Interactions. Security

In the paragraph B5 of Chapter II it has been already discussed that the implementation of this operating system has not considered the "internal security" of the system, but in the design there are all the ingredients for future extensions in this direction. A future extension also is the addition of "file management". We shall mention here two more problems which are related to the selection of the synchronization mechanism.

a. Confinement Property

During the last five years the security kernel technology has demonstrated not only that a kernel can provide

security but also that it is practical in terms of performance, functional capability, and compatibility. A successful implementation of a kernel is based on three [23] engineering principles: (1) "completeness", in that all accesses to information must go through the kernel; (2) "isolation", in that the kernel must be tamperproof; and (3) "verifiability", in that there must be a direct correspondence to the model and specification requirements.

A secure computer system will not occur as a spontaneous result of other design goals. Security must be explicitly designed in from first principles, and this is the reason why the confinement problem is discussed and has influence in the selection of the synchronization mechanism.

The major problem that has to be handled for proper system security is the "confinement property" or "* property" [24].

The "confinement property" has to prevent a process from "reading" a file with a "higher classification" or "writing" (i.e., storing or updating) a file with a "lower classification".

b. Readers/Writers Problems

Another problem closely related to the confinement problem which involves the Supervisor, is the "readers/writers" problem [25]. In order to preserve file integrity, reading and writing of a shared file cannot be allowed at the same time.

Both the confinement and readers/writers problems can be solved in one of two ways. One is mutual exclusion, a mechanism which forces a time ordering on the execution of critical regions, forces concurrent processes into a total order execution sequence. This is counterproductive to the purpose of the process structure of this implementation, which inherently allows concurrent execution of processes.

A second and relatively new method is the use of Eventcounts and Sequencers [10] to control access to critical regions. This method preserves the idea of concurrent processing to a much greater extent and also addresses the confinement property for a security kernel.

5. Synchronization Background

In order to keep processor multiplexing simple, it is desirable to have a simple interprocess communication and synchronization mechanism. Before describing the "Eventcounting" synchronization mechanism employed in the design and implementation of this operating system, it is worthwhile to discuss two generally used synchronization mechanisms, the "Semaphore" and "Block-Wakeup".

a. The "Semaphore"

In most synchronization schemes, a physical entity must be used to represent the resource. This entity is often called a "lock byte" or "semaphore". Thus, for each "shared database" (for example APT and VPM in this implementation) there should be a separate lock byte. We will use the

convention that lock byte = 0 means the resource is available, whereas lock byte = 1 means the resource is already in use.

Before operating on such a shared resource, a process must perform the following actions with no interruption:

1. Examine the value of the lock byte (either it is 0 or 1).
2. Set the lock byte to 1.
3. If the original value was 1, go back to step 1.

After the process has completed its use of the resource, it sets the lock byte to zero. Some other terms used for this operation are "Test-And-Set" instruction, "Software Lockout", "Indivisible Read-Alter-Rewrite", "Indivisible Test-And-Set" semaphore, "Spin-Lock" procedure and so on. In this design we use a built-in PL/M-86 procedure called LOCKSET, an indivisible test-and-set semaphore, to implement software locks in shared databases (APT, VPM). The hardware "bus lock" is used to make the operation indivisible. It is important to note that the lock and unlock operations do, in fact, prevent "Race Conditions".

b. "P" and "V" Operations On Counting Semaphores

A more general form of the above LOCK/UNLOCK mechanism, called the "P" and "V" operations, has been defined by Dijkstra (1968). "P" and "V" operate on the "counting semaphores" which are variables that take on integer values

(but not just 0 and 1). The mechanisms can be defined as follows:

P(S):

1. Decrement value of S (i.e., S = S-1).
2. If S is less than 0, WAIT (S).

V(S):

1. Increment value of S (i.e., S = S+1).
2. If S is less than or equal to 0, SIGNAL (S).

WAIT and SIGNAL are primitives of processor management. A WAIT (S) sets the process to the blocked state and links it to the lock byte S. Another process is then selected to run by the process scheduler. A SIGNAL (S) checks the blocked list associated with lock byte S. If there are any processes blocked waiting for S, one is selected and is set to the ready state. Then the scheduler will select a process to run.

In order to implement semaphores in the system, the processor multiplexing algorithm must be informed of all "V" operations to semaphores, and must keep track of the set of virtual processors that are waiting for each semaphore to indicate that the event has occurred.

Unfortunately semaphores have several disadvantages. First, they are limited to cases where the occurrence of an event will allow a fixed number of virtual processors to proceed out of the waiting state. (This mechanism has no "broadcast" capability). Second, because of this limitation,

the ability to proceed past a "P" operation on a semaphore automatically becomes a kind of scarce resource that can be used as a communication channel among processes that wait on the semaphore.

This latter point is quite important in a secure system design. Although communication of information is inherent in the inter-process synchronization mechanism between the virtual processor that causes an event and the virtual processors that await the occurrence of that event, there is no inherent requirement that virtual processors waiting for the same event to occur should have a communication path among themselves.

c. "Block-Wakeup"

This mechanism described in detail by Saltzer [14] is quite similar. A discussion of some problems encountered with this mechanism is presented in [15]. Reed in his thesis [15] notes:

> "If virtual processor A can wake up virtual processor B, there is no guarantee that the reason virtual processor B is waiting is the reason virtual processor A wakes B up. Virtual processor A's wakeup will then be misinterpreted by B, or ignored by B. In the first case, B will proceed under the false assumption that the event awaited happened, while in the second case, B will lose the wakeup (This is the case described by Saltzer as the "lost wakeup" problem) even though it may be meaningful to B at a later time. These problems can be serious for system security, if the wakeups are intended for a protected system operation in B's virtual processor, because a wait operation executed outside of the protected part of the system can receive inter-process synchronization signals intended for the protected part. The arrival of an inter-process synchronization signal can carry privileged system information.

An unprotected receiver may either gain unauthorized access to privileged information, or prevent it from reaching its proper destination. These occurrences cannot be prevented because B is multiplexing the meaning of his wakeup-waiting switch, and so must allow A to wake him up at all times, even though B waits for A's event only sometimes".

For these reasons, along with the need to deal with synchronization in "distributed" systems, Kanodia and Reed [10] have designed an inter-process synchronization mechanism that is in some sense more general than either semaphores or block-wakeup, and uses "Eventcounts" and "Sequencers". We shall discuss eventcounts and sequencers later on in this Chapter.

6. Communication and Synchronization In This Implementation

a. Introduction

The design of this operating system supports multiprogramming and multiprocessing. Multiprogramming is used to improve system efficiency and to create a virtual environment which frees the remainder of the operating system from a dependence on the physical processor configuration. On the other hand the process structure provides the essentials for parallel (concurrent) processing. In a multiprocessor environment concurrent processing can provide faster completion of a job. Using n processors working on the same job and each of them doing separate tasks (after a suitable partitioning of the job), the overall time required to run the job can be reduced, frequently by a factor n.

The above discussion provides some of the major reasons why this system is designed to support concurrent processing on multiple processors. In addition, the existence of multiple physical processors gave rise to the need for the design of processor multiplexing to be done in two-levels. The Traffic Controller that multiplexes processes among virtual processors and the Inner Traffic Controller that multiplexes physical processors among a fixed larger set of virtual processors.

Since this system will also be used to support real-time processing, a pre-emption mechanism is provided to facilitate preemptive scheduling.

The above process multiplexing, processor multiplexing, and preemptive scheduling require the following support in communication and synchronization:

(1) Inter-process communication and synchronization, at the Traffic Controller level.

(2) Inter-virtual processor communication and synchronization at the Inner Traffic Controller level.

(3) Inter-real processor communication needed to support the preemptive scheduling.

b. Inter-Process Communication and Synchronization

For concurrent processing, a job composed of sequential and non-sequential tasks, is explicitly divided (partitioned) into an appropriate structure of processes that can run concurrently. There is the possibility that

after the partitioning the resulting processes must interact (need cooperation).

It is the responsibility of the operating system to provide mechanisms for communication and synchronization between cooperating processes. There are two different kinds of interaction that processes must be able to achieve.

First there must exist a way for processes to exchange data. This mode of communication is called "inter-process communication". In a computer system that allows sharing of memory segments between processes (in our case shared segments will reside in the "global" memory board), there is no need for a special inter-process communication facility to be built into the processor multiplexing algorithm. Shared memory segments provide an extremely high bandwidth data communication channel between the processes sharing these segments. Any protocol for inter-process communication can be established by the processes using the shared segments. Therefore the inter-process communication will be handled outside of the scope of this thesis. The responsibility is left to the user of the operating system, since it is dependent on the specific application program.

Secondly there must exist a way for processes to wait for data prepared by other processes and for processes that prepare such data to signal that this data is available. This interaction is different than communication of data and is called "inter-process synchronization". Together they are

called "inter-process communication and synchronization". Another term for inter-process synchronization is "inter-process control communication" since the effect of such communication is purely to reenable a waiting control point.

The actual coordination is realized inside the kernel by the use of "shared writable" segments and is used for controlling the execution of processes and coordinating the sharing of data.

The synchronization between processes is "visible" to the user and is supported by the TC$AWAIT and TC$ADVANCE that are kernel calls to the Traffic Controller level. We have already discussed the basics of these two synchronization primitives in paragraphs C6a, C7a and D of Chapter II. The details are described in the corresponding modules of the Traffic Controller in this chapter.

The inter-process synchronization is intimately related to the structure of the "processor multiplexing mechanism". The ability of a process to indicate that it does not need virtual processor resources until a particular "event" happens is basic to the economic advantage of process multiplexing among virtual processors.

c. Inter-virtual Processor Communication and Synchronization

The ability of a virtual processor to indicate that it does not need real processor resources until a particular "event" happens is, similarly, basic to the economic advantage of virtual processors multiplexing among real

processors. Otherwise if a dedicated real processor is actually available for each virtual processor, then the "busy-waiting" would be an adequate mechanism for synchronization. ("Busy-waiting" is repeatedly testing the state of a shared memory word in a loop).

If for example, a user process calls upon some system service, such as a disk I/O or an I/O for a real-time sensor, it must wait for that service to be completed before it can proceed. (The performance of a system service is, in this case, considered to be part of the requesting process). However, the service may actually be supported by another virtual processor. To control this interaction, the Inner Traffic Controller that multiplexes physical processors among virtual processors, provides the required inter-virtual processor communication and synchronization mechanism using the primitives ITC$AWAIT and ITC$ADVANCE.

We have already discussed the basics of these two synchronization primitives in paragraphs C6b, C7b and D of Chapter II. The details are described in the corresponding modules of the Inner Traffic Controller in this chapter.

This inter-virtual processor synchronization is "invisible" to the user, and is used by the operating system for the management of physical resources. This mechanism provides the solution to a difficult problem: "the synchronization" that will be faced later on, when the

"Memory Management" and "I/O Management" are added to the operating system.

d. Inter-Real Processor Communication

To support real-time processing we need the preemptive scheduling. Since we are working in a multiple-processor environment the operating system has to support an inter-real processors communication mechanism, which is of course related to the inter-virtual processors synchronization mechanism. It will be explained by the two examples below.

It is important to note that the preemptive scheduling mechanism is completely distinct from the synchronization mechanism and its purpose is to cause the "immediate attention" of a real processor when it is needed for real time applications.

The TC$ADVANCE and ITC$ADVANCE modules provide a "broadcast" capability. Let us examine first the case of TC$ADVANCE. When an application (user) process calls the TC$ADVANCE, the result is an incrementing by one of the associated event's current value. This change of the event's value is "broadcast" to all processes that are awaiting this value for the specific event. We have to remember here that the operating system is distributed to each Real Processor and also that each real processor in this implementation possesses four virtual processors. If a process waiting for the above specific event is bound to a Virtual Processor which belongs to another real processor, then there is no way

to signal that virtual processor to inform it of the occurrence of this event. Similarly, if during the physical resources management (for example I/O management) the ITC$ADVANCE is invoked by the operating system (this is "invisible" to the user), it results again in an incrementating by one of the associated event's current value (now in the Inner Traffic Controller level). If this change has to be "broadcast" to a Virtual Processor awaiting this event and the Virtual Processor belongs to another Real Processor, then again there is no way to inform that Virtual Processor of the occurrence of the specific event.

To facilitate the inter-real processor communication, we employ the hardware interrupt. Similar to most microcomputers, the 8086 microprocessor does not have the capability to send hardware interrupts destined for other devices (here the devices of interest are other CPU's). To solve the problem we have suitably configured the hardware using the on board (8086 microcomputer) hardware chips, 8259A Programmable Interrupt Controller and 8255A Programmable Peripheral Interface and the Multibus interface.

This configuration is discussed in detail in paragraph G of this chapter.

e. Events, Eventcounts, and Sequencers

The ability to synchronize the execution of processes throughout the system (irrespective of which microcomputer they are loaded on) is the cornerstone of the power

and flexibility of this system. To accomplish this, process synchronization is based on the notion of "events".

An "event" is anything that one considers significant and can direct, in some fashion, the computer to respond to. For example events of interest are: the completion of a program, a buffer becomes full or empty, a printer is ready, a process in execution on a VP reaches a control point defined by the user. More generally, the events can represent virtually anything of interest to the programmer.

When an event occurs, the computer recognizes that it is to respond in some specified manner.

"Eventcounts" and "sequencers" allow processes to synchronize with each other somewhat indirectly. To synchronize directly, a process would have to somehow identify the other processes with which it is synchronizing (viz., explicitly signal a process by name). This would require the naming of individual processes or some similar identification scheme.

Rather than using a process naming scheme, the individual processes "agree", in a sense, to cooperate by using a common set of memory objects called eventcounts and sequencers. In this way, even though the processes must know the names of the eventcounts and sequencers that they use, they are not required to know anything at all about each other's identities. In fact, a process need not even know how many other processes will be synchronizing with it. This offers some advantages in parallel processing. Processes that

synchronize with eventcounts do not have to know how many other processes will also use the same eventcounts. This means that fewer coding changes will be required when, for example, a single process is partitioned into several processes all executing in parallel. All of the "new" processes will synchronize on the same eventcount so that no changes are required in the process that originally synchronized with the single process.

Eventcounts are used to keep track of the occurrence of specific events. They are managed for the user by the system. Sequencers can be used to impose a linear order on the occurrence of events. They are thus used with eventcounts to provide for mutual exclusion.

f. Eventcounts

"Eventcounts" are used in this implementation to allow processes to arbitrate access to shared resources. An eventcount is defined by Reed [10] as: "An eventcount is an object in the system that represents a class of events that will eventually occur". Each eventcount represents a distinct class of events. This class of events is ordered so that by the time event N occurs all events numbered from 0 to N-1 will have occurred. Consequently, the set of events that have occurred at any particular time can be represented by the number of the last event to occur. This number is known as the "current value" of the eventcount.

An eventcount is associated with some type of event of interest, e.g., occurrence of a real-time interrupt, a data segment being read or written into, etc. Eventcounts are implemented as sets of positive integers from 0 to infinity (the current limit in this implementation is actually 65,536 using PL/M-86 "word" variables which is "adequate" for the applications anticipated) and are used to keep track of the total number of such events that have occurred.

The eventcount synchronization mechanism has the useful property that two virtual processors waiting for events in the same class (thus recorded in the same eventcount) do not have an inherent intercommunication path. The enabling of one virtual processor to proceed does not automatically disable any other virtual processors from proceeding and allows broadcasting events to multiple virtual processors. This is a function not easily achieved using semaphores. Consequently, this mechanism is more desirable for use in a secure system to address the "confinement property". Further, the implementation of eventcounts is not inherently more difficult than that of semaphores.

There are three operations which may be performed on eventcounts, as follows:

(1) "Read" Operation. The current value of an eventcount may be obtained by the READ operation. This operation returns the present value of the eventcount as a "positive

integer" n. From this value, one may infer that events 0 to n have already occurred. TC$READ (Traffic Controller READ) in the present implementation is a function call in the Traffic Controller Level available ("visible") to the user via the "GATE" so that it will provide him the capability to obtain, the current value of the eventcount of interest specified in the call. Details will be discussed in the corresponding module of the Traffic Controller later in this chapter.

(2) "AWAIT" Operation. Allows the calling subject to await a particular event in the class associated with the eventcount. This operation requires that the event name and the awaited eventcount value be specified. Particularly in the present implementation there are two procedures as follows:

TC$AWAIT (Traffic Controller AWAIT) is an inter-process synchronization primitive. Allows a process (the "calling" process) to suspend its own execution (enter the "blocked" state) until the event specified in the input argument (by name and value) has occurred, viz., the eventcount reaches the specified awaited value. The result is that the process will "give away" the virtual processor to which it is bound. The effect of this operation is similar as the conventional Saltzer's "Block" operation or Dijkstra's "P" operator (on counting semaphores).

TC$AWAIT is a procedure in the Traffic Controller Level "visible" to the user via the "GATE". Details will be discussed in the corresponding module of the Traffic Controller later in this chapter.

ITC$AWAIT (Inner Traffic Controller AWAIT), is an inter-virtual processor synchronization primitive. It suspends the execution of the "running" virtual processor (setting its state to "waiting") until the event specified (by name and value) in the input argument has occurred, viz., the eventcount reaches the specified awaited value. This synchronization primitive is used by the Inner Traffic Controller for the management of system resources. ITC$AWAIT, is "invisible" to the user, and is used only by the operating system. Details will be discussed in the corresponding module of the ITC later in this chapter.

TC$AWAIT/ITC$AWAIT will prevent the process/ virtual processor respectively from proceeding until the current value of the eventcount reaches the awaited event value specified in the procedure's call.

(3) "ADVANCE" Operation. This operation informs the processor multiplexing mechanism of the new value of the advanced eventcount and requires that the event name be specified as an argument. Particularly in this implementation there are two procedures as follows:

TC$ADVANCE (Traffic Controller ADVANCE) is an inter-process synchronization primitive. A TC$ADVANCE operation is performed by a process when an event has occurred. It increments the current value of the specified eventcount by one to reflect the occurrence of the event. This has the effect of signalling the event's occurrence to other processes

which were waiting for it by virtue of having previously performed an AWAIT operation on that event. The effect of an ADVANCE operation is essentially the same as a Saltzer's Wakeup operation of Dijkstra's "V" operator (on counting semaphores).

The eventcount signalling mechanism has an "automatic broadcast effect" which offers an advantage in parallel processing. This broadcast capability allows the "simultaneous signalling" of several processes which otherwise would have to be signalled "sequentially".

TC$ADVANCE is a procedure in the Traffic Controller Level "visible" to the user via the "GATE". TC$ADVANCE is also in this implementation responsible for the "preemptive scheduling". Details will be discussed in the corresponding module of the Traffic Controller later in this chapter.

ITC$ADVANCE (Inner Traffic Controller ADVANCE), is an inter-virtual processor synchronization primitive. Signals that the specified in the call event (event's name is the input argument) has occurred by advancing (incrementing by one) the value of the associated eventcount. This eventcount signalling mechanism has also an "automatic broadcast" effect which offers an advantage in parallel processing. All the virtual processors awaiting the occurrence of this specific event are informed. ITC$ADVANCE is a procedure in the Inner Traffic Controller

Level "invisible" to the user and is used only by the operating system for the management of system's resources. Details will be discussed in the corresponding module of the ITC later in this chapter.

g. Sequencers

There are many situations where accesses to shared resources must be totally ordered. Eventcounts alone are not sufficient to accomplish this. To provide the capability for mutual exclusion, another type of object called a "sequencer" [10] is employed. A sequencer is implemented as a positive integer ranging in value from 0 to infinity (as with eventcounts, the current limit in this implementation is 65,536). However, a sequencer is used to provide total order to the occurrence of events.

A sequencer is also necessary to solve the confinement and readers/writers problems. Some synchronization problems require arbitration, e.g., two write accesses to the same segment. Eventcounts alone as already discussed do not have the ability to discriminate between two events that happen in an uncontrolled (i.e., concurrent) manner.

Initially a sequencer has a value of 0. The value increases by one each time a "TICKET" operation is performed on it. TICKET is the only operation defined on a sequencer. TICKET returns a unique monotonically increasing value with each call. It is similar to getting a ticket and waiting to be served at a restaurant. Two uses of TICKET will return

two different values corresponding to the "relative time" of call. Thus, a set of events can be totally ordered by using the TICKET operation. Details about TICKET operation will be discussed in the corresponding module of the TC later in this chapter.

G. INTERRUPT STRUCTURE

1. Introduction

The operating system has to control a multiple-processor environment. This generates the need of some method of communication between physical processors. This need is satisfied by an ability to generate hardware interrupts between the physical processors. The interrupts are used for the implementation of "preemptive scheduling". INTEL's 8086 microprocessor, as most microprocessors, doesn't possess the capability to directly generate interrupts destined for other devices (the devices of interest here are other processors). We provide that capability by suitably configuring the hardware and using some software control. Note that only a "single" interrupt line is actually used to implement system-wide preempt interrupts. This is the only hardware configuration adaptation to facilitate the operating system and we are going to describe it in detail.

The system's interrupt structure is managed by the Inner Traffic Controller. In particular, a physical system interrupt is transformed into a synchronization signal to a

waiting virtual processor. This structure is particularly important for the support of real-time processing and note that this is completely distinct from inter-process synchronization and communication.

To implement this desired configuration we use the 8259A PIC (Programmable Interrupt Controller) and 8255A PPI (Programmable Peripheral Interface), both on board on the 86/12A microcomputer.

The 8086 instructions support two types of interrupts, external and internal (or "trap"). An external interrupt is initiated by some peripheral asserting an interrupt request to the 8086 in the hardware. An internal interrupt is one initiated by the software the 8086 is executing. An interrupt represents a transfer of program execution control. The type of transfer used in the 8086 is called a vectored interrupt. An interrupt vector represents an address of a procedure which services the interrupt.

In the 8086 all interrupts (both external and internal) perform a transfer by pushing the flag registers onto the stack (as in PUSHF), and then performing an indirect call (of the intersegment variety) through an element of an interrupt vector located at absolute memory locations 0 through 3FFH. Each vector is a four byte element with the first two bytes containing the offset of a procedure (or label) and the second two bytes containing the paragraph number of the segment containing the procedure (or label). There are 256

possible interrupt vectors. Within the 8086 assembly language, each vector is given a number from 0 through 255. Interrupts 0 through 4 (0-13H) currently have the dedicated hardware functions as defined on Figure 29 below (the dedication has been made by INTEL Corporation).

Interrupt #	Location	Function
0	0-03H	divide by zero
1	04H-07H	single step
2	08H-0BH	non-maskable interrupt
3	0CH-0FH	one byte interrupt instruction (INT 3)
4	10H-13H	interrupt on overflow

FIGURE 29. INTERRUPTS 0 to 4.

There are three interrupt transfer operations provided:

- INT pushes the flag registers, clears the TF (Trap Flag) and IF (Interrupt Flag) flags, and transfers control with an indirect call through any of the 256 vector elements, i.e., INT 24 will do an indirect call through interrupt vector 24 (location 96). A one byte form of this instruction is available for interrupt type 3, INT 3. We use INT instruction for the implementation of the "GATE".
- INTO pushes the flag registers, clears the TF and IF flags and transfers control through vector element 4

if the OF flag is set (interrupt on overflow). If the OF flag is cleared, then no operation takes place.

- IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag registers. This instruction is used several times for the implementation of the operating system.

For external interrupts, the peripheral device will request an interrupt from the 8086. When the 8086 grants the interrupt, the device will supply a byte value on the data bus which represents the type or number of the interrupt i.e., 0 through 255. The 8086 will read this value and then execute the interrupt through the vector.

2. Hardware Interrupts

The 8086 CPU includes two hardware interrupt inputs, NMI and INTR, classified as non-maskable and maskable, respectively.

a. Non-Maskable Interrupt (NMI)

The NIM input has the higher priority of the two interrupt inputs. A low-to-high transition on the NMI input will be serviced at the end of the current instruction or between whole moves of a block-type instruction. Worst-case response to NMI is during a multiply, divide, or variable shift instruction.

When the NMI input goes active, the CPU performs the following:

(1) Pushes the Flag registers onto the stack (same as a PUSHF instruction).

(2) If not already clear, clears the Interrupt Flag (same as a CLI instruction). This disables maskable interrupts.

(3) Transfers control with an indirect call through vector location 00008.

The NMI input is intended only for catastrophic error handling such as a system power failure. Upon completion of the service routine, the CPU automatically restores the flags and returns to the main program.

b. Maskable Interrupt (INTR)

The INTR input has the lower priority of the two interrupt inputs. A high level on the INTR input will be serviced at the end of the current instruction or at the end of the whole move for a block-type instruction.

When INTR goes active, the CPU performs the following (assuming the Interrupt Flag is set):

(1) Issues two acknowledge signals. Upon receipt of the second acknowledge signal, the interrupting device (master or slave PIC) will respond with a one-byte interrupt identifier.

(2) Pushes the Flag registers onto the stack (same as a PUSHF instruction).

(3) Clears the Interrupt Flag thereby disabling further maskable interrupts.

(4) Multiplies by four (4) the binary value (X) contained in the one-byte identifier from the interrupting device.

(5) Transfers control with an indirect call through location 4X.

Upon completion of the service routine, the CPU automatically restores its flags and returns to the main program.

3. 8259A PIC (Programmable Interrupt Controller)

The on board 8259A PIC functions as an overall manager in an interrupt-driven system environment. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest importance (priority), ascertains whether the incoming request has a higher priority value than the level currently being serviced and may issue an interrupt to the CPU based on this determination.

The on board master 8259A PIC handles up to eight vectored priority interrupts and has the capability of expanding the number of priority interrupts by cascading one or more of its interrupt input lines with slave 8259A PIC's. Note that slave PIC's are not used in this implementation.

The basic functions of the PIC are to (1) resolve the priority of interrupt requests, (2) issue a single interrupt request to the CPU based on that priority, and (3) send the CPU a vectored restart address for servicing the interrupting device.

a. Interrupt Priority Modes

The PIC can be programmed to operate in one of the following modes:

(1) Nested Mode

(2) Fully Nested Mode

(3) Automatic Rotating Mode

(4) Specific Rotating Mode

(5) Special Mask Mode

(6) Poll Mode

In this design the Nested Mode is used and is described in the next paragraph.

b. Nested Mode

In this mode the PIC input signals are assigned a priority from 0 through 7. The PIC operates in this mode unless specifically programmed otherwise. Interrupt IR0 has the highest priority and IR7 has the lowest priority. When an interrupt is acknowledged, the highest priority request is available to the CPU. Lower priority interrupts are inhibited, higher priority interrupts will be able to generate an interrupt that will be acknowledged, if the CPU has enabled its own interrupt input through software. The End-Of-Interrupt (EOI) command from the CPU is required to reset the PIC for the next interrupt.

Details for the remaining modes are described in Reference [2].

c. Status Read

Interrupt request inputs are handled by the following three internal PIC registers:

(1) Interrupt Request Register (IRR) which stores all interrupt levels that are requesting service.

(2) In-Service Register (ISR) which stores all interrupt levels that are being serviced.

(3) Interrupt Mask Register (IMR) which stores the interrupt request lines which are masked.

These registers can be read by writing a suitable command word and then performing a read operation.

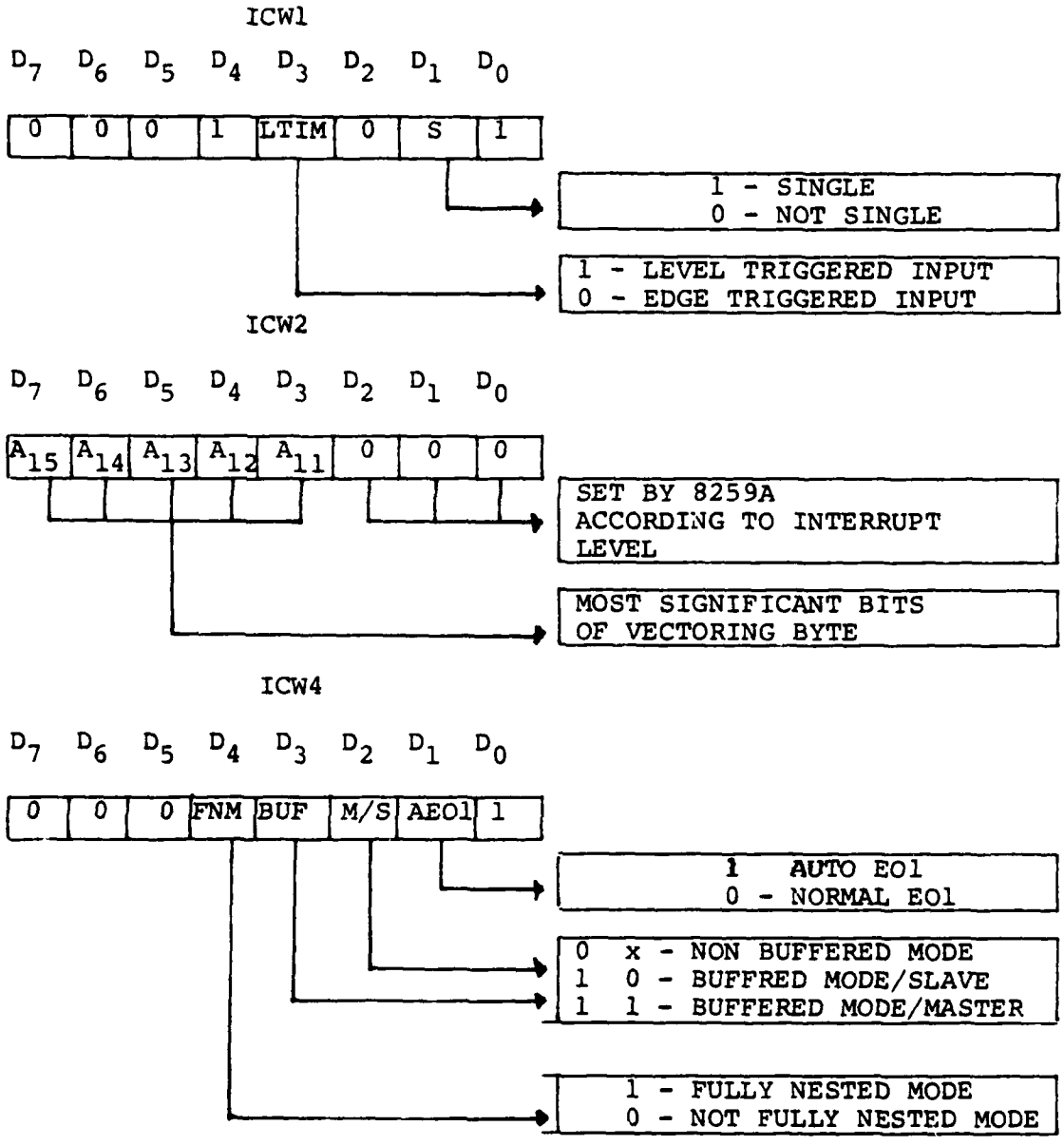
d. Initialization Command Words

The on board master PIC and each slave PIC requires a separate initialization sequence to work in a particular mode. The initialization sequence requires three Initialization Command Words (ICW's) for a signle PIC system and requires four ICW's for a master PIC with one to eight slaves. The ICW formats are shown in Figure 30. Since no slave PIC's are used we shall describe below only the initialization command words needed to initialize the on board PIC.

The First Initialization Command Word (ICW1), which is required in all modes of operation consits of the following:

(1) Bits 0 and 4 are both 1's and identify the word as ICW1 for an 8086 CPU operation.

(2) Bit 1 denotes whether or not the PIC is employed in a multiple PIC configuration. For a single master PIC configuration (no slaves) bit 1=1; for a master with one or more slaves bit 1=0. Note that bit 1=0 only when programming a slave PIC.



NOTE: X INDICATED "DON'T CARE"

FIGURE 30. PIC INITIALIZATION COMMAND WORD FORMATS

(3) Bit 3 establishes whether the interrupts are requested by a positive-true level input or requested by a low-to-high input. This applies to all input requests handled by the PIC. In other words, if bit 3=1, a low-to-high transition is required to request an interrupt on any of the eight levels handled by the PIC.

The second Initialization Command Word (ICW2) represents the vectoring byte (identifier) and is required by the 8086 CPU during interrupt processing. ICW2 consists of the following:

(1) Bits D3-D7 (A11-A15) represent the five most significant bits of the vector byte. These are supplied by the programmer.

(2) Bits D0-D2 represent the interrupt level requesting service. These bits are provided by the 8259A during interrupt processing. These bits should be programmed as 0's when initializing the PIC.

Note that the 8086 CPU multiplies the vector byte by four. This value is then used by the CPU as the vector address.

Figure 31 lists the vector byte contents for interrupts IR0-IR7.

	D7	D6	D5	D4	D3	D2	D1	D0
IR7	A15	A14	A13	A12	A11	1	1	1
IR6	A15	A14	A13	A12	A11	1	1	0
IR5	A15	A14	A13	A12	A11	1	0	1
IR4	A15	A14	A13	A12	A11	1	0	0
IR3	A15	A14	A13	A11	A10	0	1	1
IR2	A15	A14	A13	A12	A11	0	1	0
IR1	A15	A14	A13	A12	A11	0	0	1
IR0	A15	A14	A13	A12	A11	0	0	0

FIGURE 31. INTERRUPT VECTOR BYTE.

It is important here to notice that the monitor of each microcomputer [21] initializes the PIC. For testing this hardware configuration the interrupt line 4 is connected and the interrupt vector byte (of Figure 31) is initialized (just for this kernel program) to 40H. Note that the three LSB bits (D0, D1, D2) are always initialized to 0. For the specific initialization, bit D6=1 and all the rest are 0. Since the interrupt line 4 is connected, the PIC upon receiving an interrupt resolves the priority and sets the bits D2=1, D1=0, D0=0 (D2D1D0=100=4). Therefore, the interrupt vector byte is set to 40H+4=44H. The 8086 CPU multiplies the interrupt vector byte by 4 and the resulting value, 110H, is the vector address. The CPU will transfer control to this address to execute the interrupt service routine corresponding to the interrupt 4. A pointer (four bytes) pointing to the starting point of the interrupt service routine must be located in the physical absolute address (110H) corresponding to the received interrupt.

Since both the monitor and the kernel initialize the PIC there exists a probability of conflict as follows: If the first 100 bytes of local RAM memory of every microcomputer will be displayed using the monitor's display command, as in Figure 32, then we can see that the monitor uses 12 bytes (04 to 0F). Also 32 bytes are occupied (80H to 9FH) and these are pointers to a single entry point (pointer 6C 06 00 FE is repeated 8 times). If the interrupt

```
0000:0000  00 00 00 00 06 04 00 FE DB 05 00 FE DB 05 00 FE
0000:0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:0080  6C 06 00 FE 6C 06 00 FE 6C 06 00 FE 6C 06 00 FE
0000:0090  6C 06 00 FE 6C 06 00 FE 6C 06 00 FE 6C 06 00 FE
0000:00A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:00B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:00C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:00D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:00E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000:00F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

FIGURE 32. DISPLAY OF THE FIRST 100H BYTES

vector address, after the PIC initialization, happens to be between 80H and 9FH the interrupt service routine pointer is overwritten by the monitor. The solution is to substitute (using monitors' "S" command) the service routine vector in place of the monitor's interrupt service routine vector. For example if we initialize the interrupt vector byte of Figure 31 with 20H and use interrupt line 4, then after CPU's multiplication by 4, the resulting vector address is 24H*4=90H. Before execution we have to substitute the four bytes 90H to 93H with the interrupt service routine pointer.

If we avoid the area (from 80H to 9FH) then there is no problem. Also when the operating system (instead of the monitor) will be the permanent resident of ROM, this problem will not exist. (See also Anderson's thesis [19]).

Now the PIC initialization is continued.

The third initialization command word, ICW3, is not required for this implementation since we do not use slave PIC's.

The fourth Initialization Command Word (ICW4), which is required for all modes of operation, consists of the following:

(1) Bit D0 is a 1 to identify that the word is for an 8086 CPU.

(2) Bit D1 (AEOI) programs the end-of-interrupt function. Code bit 1=1 if an EOI is to be automatically executed (hardware). Code Bit 1=0 if an EOI command is to be generated by software before returning from the service routine.

(3) Bit D2 (M/S) specifies if ICW4 is addressed to a master PIC or a slave PIC. For example, code bit 2=1 in ICW4 for the master PIC. If bit D3 (BUF) is zero, bit D2 has no function.

(4) Bit D3 (BUF) specifies whether the 8259A is operating in the buffered or nonbuffered mode. For example, code bit 3=1 for buffered mode.

The master PIC in an iSBC 86/12A, with or without slaves, must be operated in the buffered mode.

(5) Bit D4 (FNM) programs the nested or fully nested mode.

In summary, three ICW's are required to initialize the on board PIC in this implementation, ICW1, ICW2 and ICW4.

e. Operation Command Words

After being initialized, the master and slave PIC's can be programmed at any time for various operating modes. The Operation Command Word (OCW) formats are shown in Figure3-15 of Reference [2]. The format of the only one operation command word used in this implementation (OCW1) is shown in Figure 33.

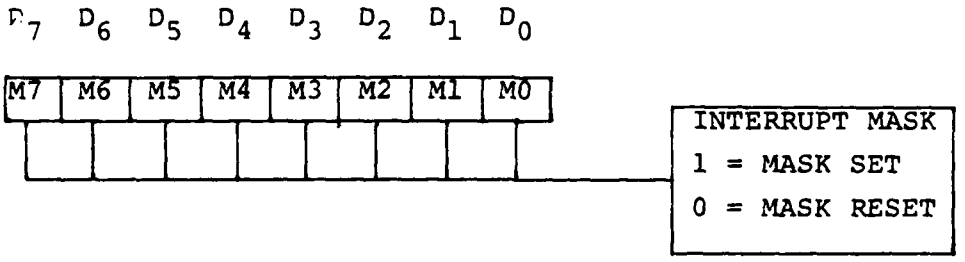


FIGURE 33. OPERATION COMMAND WORD #1, (OCW 1)

f. Addressing

The master PIC uses Port 00C0 or 00C2 to write initialization and operation command words and Port 00C4 or 00C6 to read status, poll and mask bytes. Addresses for the specific functions are provided in Reference [2].

g. Initialization

To initialize the PIC the following steps must be followed:

1. Disable system interrupts by executing a CLI (Clear Interrupt Flag) instruction.

2. Initialize master PIC by writing ICW's in the following sequence:

   Write ICW1 to Port 00C0 and ICW2 to Port 00C2.

   Write ICW4 to Port 00C2.

3. Enable system interrupts by executing an STI (Set Interrupt Flag) instruction.

h. Operation

After initialization, the master PIC and slave PIC's can independently be programmed at any time by an Operation Command Word (OCW) for the following operations:

(1) Auto-rotating priority.

(2) Specific rotating priority.

(3) Status read of Interrupt Request Register (IRR).

(4) Status read of In-Service Register (ISR).

(5) Interrupt mask bits are set, reset, or read.

(6) Special mask mode set or reset.

The details of these Operation Command Words are described in Reference [2]. In this implementation, only the OCW1 is used which has already been described.

4. 8255A PPI (Programmable Peripheral Interface)

The three parallel I/O ports interfaced to connector J1 of the 86/12A *microcomputer* are controlled by an INTEL 8255 Programmable Peripheral Interface chip. Port A includes bidirectional data buffers and Ports B and C include IC sockets for installation of either input terminators or output drivers depending on the user's application.

Default jumpers set the Port A bidirectional data buffers to the output mode. Optional jumpers allow the bidirectional data buffers to be set to the input mode or allow any one of the eight Port C bits to selectively set the Port A bidirectional data buffers to the input or output mode.

Reference [2] lists the various operating modes for the three PPI parallel I/O ports. Note that Port A (00C8) can be operated in Modes 0, 1, or 2; Port B (00CA) can be operated in Mode 0 or 1; Port C (00CC) can be operated in Mode 0.

a. Control Word Format

The control word format shown in Figure 34 is used to initialize the PPI in order to define the operating mode of the three ports. Note that the ports are separated into two groups. Group A (control word bits 3 through 6)

defines the operating mode for Port A (00C8) and the upper four bits of Port C (00CC). Group B (control word bits 0 through 2) defines the operating mode for Port B (00CA) and the lower four bits of Port C (00CC). Bit 7 of the control word controls the mode set flag.

b. Addressing

The PPI uses four consecutive even addresses (00C8 through 00CE) for data transfer, obtaining the status and control of the PPI at Port C (00CC).

c. Initialization

To initialize the PPI, a control word is written to the port address 00CE. In Figure 34, an example is given for the PPI initialization. In this example, the control word is 92H. This initializes the PPI as follows:

(1) Mode Set Flag active

(2) Port A (00C8) set to Mode 0 Input

(3) Port C (00CC) upper set to Mode 0 Output

(4) Port B (00CA) set to Mode 0 Input

(5) Port C (00CC) lower set to Mode 0 Output

d. Operation

After the PPI has been initialized, the operation is completed by simply performing a read or a write to the appropriate port.

5. The Actual Configuration

a. Hardware Connections

The hardware connections to implement this hardware adaptation are marked with special comments in the following

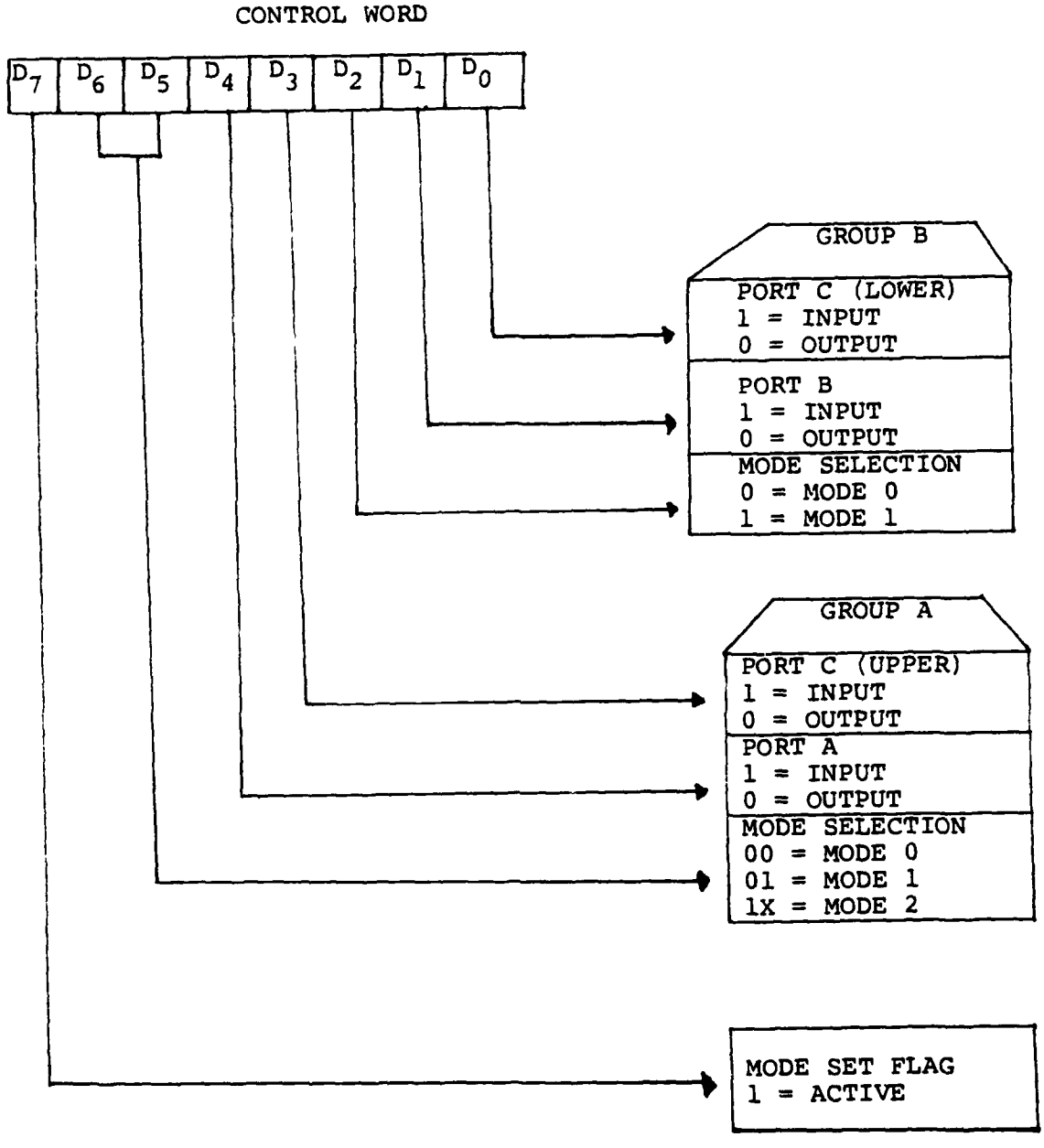


FIGURE 34. PPI CONTROL WORD FORMAT

Figures 36 and 37. In Figure 36 (This is the Figure 5-2, sheet 9 of 11, of Reference [2]) pin E9 is connected with pin E14. This connection will connect PC7 (bit 7, e.g., the MSB of Port "C") to the BUS INTR OUT. Port "C" and BUS INTR OUT line are shown on Figure 35.

In Figure 37, (This is the Figure 5-2, sheet 8 of 11 of Reference [2]) pin E137 is connected with pin E142. This connection will connect INTR 4 (interrupt 4 line) to the BUS INTR OUT. Then pin E69 is connected with pin E77. This connection will connect BUS INTR OUT to the IR4 (interrupt 4) input of the 8259A PIC (Programmable Interrupt Controller), via the Interrupt Matrix. INTR4, IR4 and Interrupt Matrix are shown in Figure 35.

With the above three jumpers, we connected the MSB (bit 7) of Port "C" (of 8255A Programmable Peripheral Interface) with IR4 (interrupt 4 input of the 8259A PIC). These connections have to be made on every 86/12A microcomputer in the system.

We have to note here that interrupt line 4 is selected arbitrarily. It is possible to connect a different line or to connect parallel Port "A" or "B". We selected "C" in order not to interfere with the operations of the data ports "A" and "B".

b. Software Control

In order to receive an interrupt the 8259A PIC has to detect a "Low to High" transition in the corresponding input

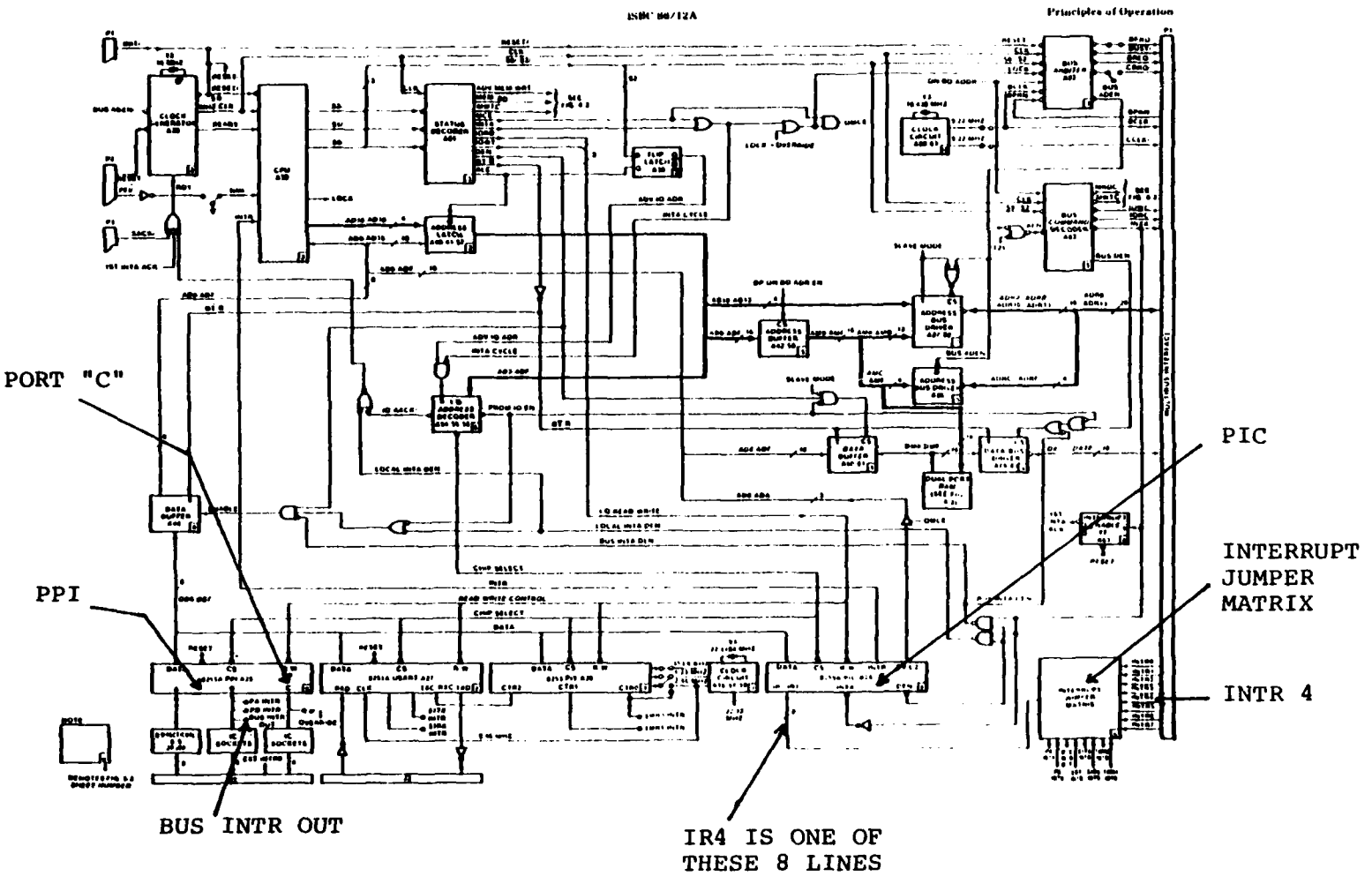


FIGURE 35. iSBC 86/12A INPUT/OUTPUT AND INTERRUPT SIMPLIFIED LOGIC DIAGRAM

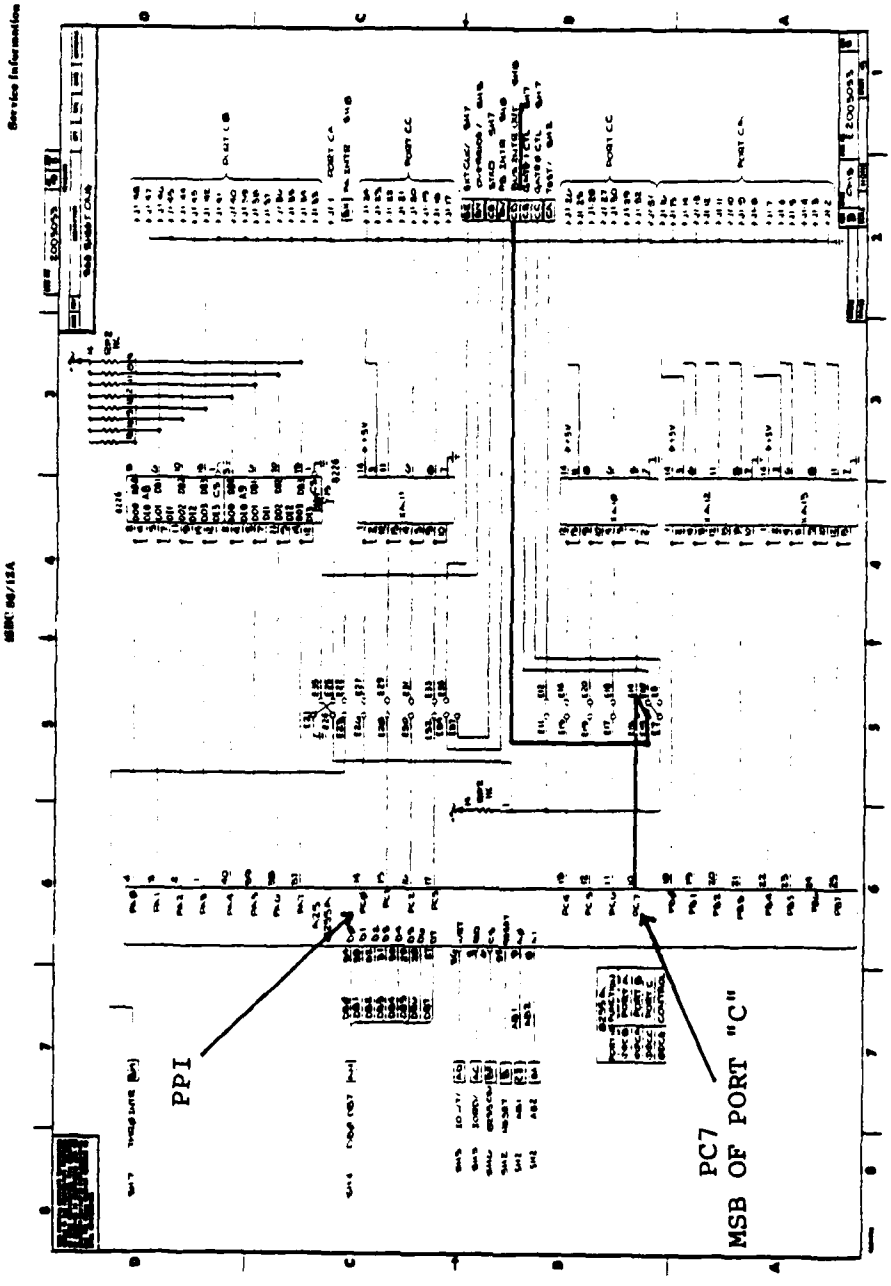


FIGURE 36. 8255A PPI, BUS INTERRUPT OUT SCHEMATIC DIAGRAM

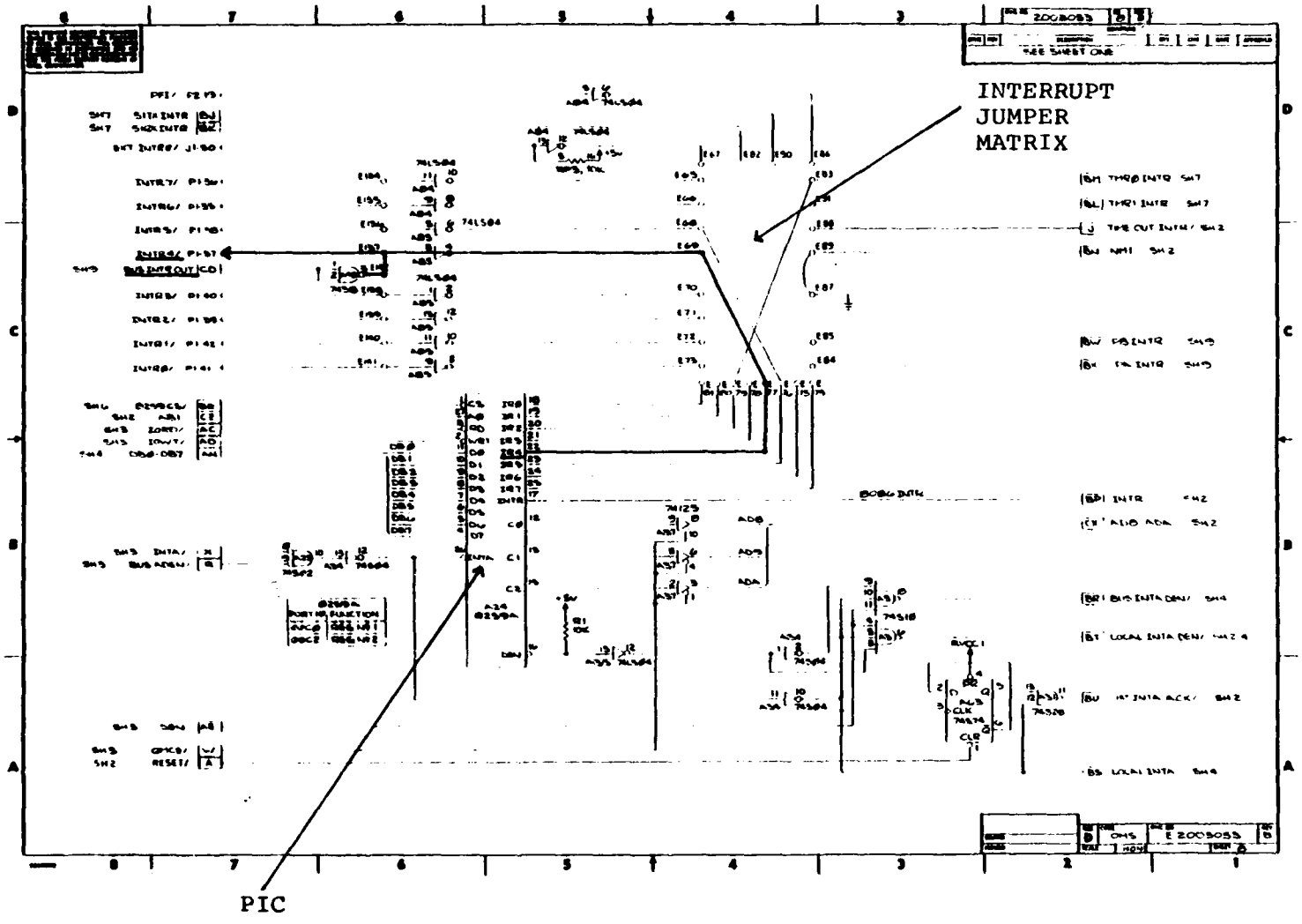


FIGURE 37. 8259A PIC, INTERRUPT MATRIX SCHEMATIC DIAGRAM

(IR 4 in our case). Since we already have connected PC7 (MSB of Port "C") with the interrupt line 4, we only need to "Reset-Set" that MSB by writing a byte into the Parallel Port "C" (specifically to port address "00CC").

Since the Port "C" is an eight bit port, to reset the MSB (PC7) we can write to Port "00CC" any number from 0 to 79H (MSB equal 0). To set the MSB we can write any number from 80H to OFFH (MSB equal 1).

We also use a "global" array of flags, called in the implementation HDWINTFLAG (Hardware Interrupt Flag). HDWINTFLAG (n) corresponds to the processor whose identification number (CPU$NUMBER) equals n. Since this flag array is global, each physical processor can access the flag of any other processor in the system.

This way we establish an effective and simple design and implementation of the "inter-physical processor communication" using just "one" hardware interrupt line. The algorithm is shown in Figure 38. When a processor #n needs to preempt another processor #m, then it first set its corresponding flag, e.g., HDWINTFLAG(m) = TRUE and afterwards sends a hardware interrupt by writing to Port address "00CC" first a zero, then an 80H and finally a zero. This way processor #n generates the "Low to High" transition at the interrupt 4 input (IR 4) of the 8259A PIC of "every" 86/12A microcomputer (including itself) in the system. Then every processor jumps to the interrupt handler that first

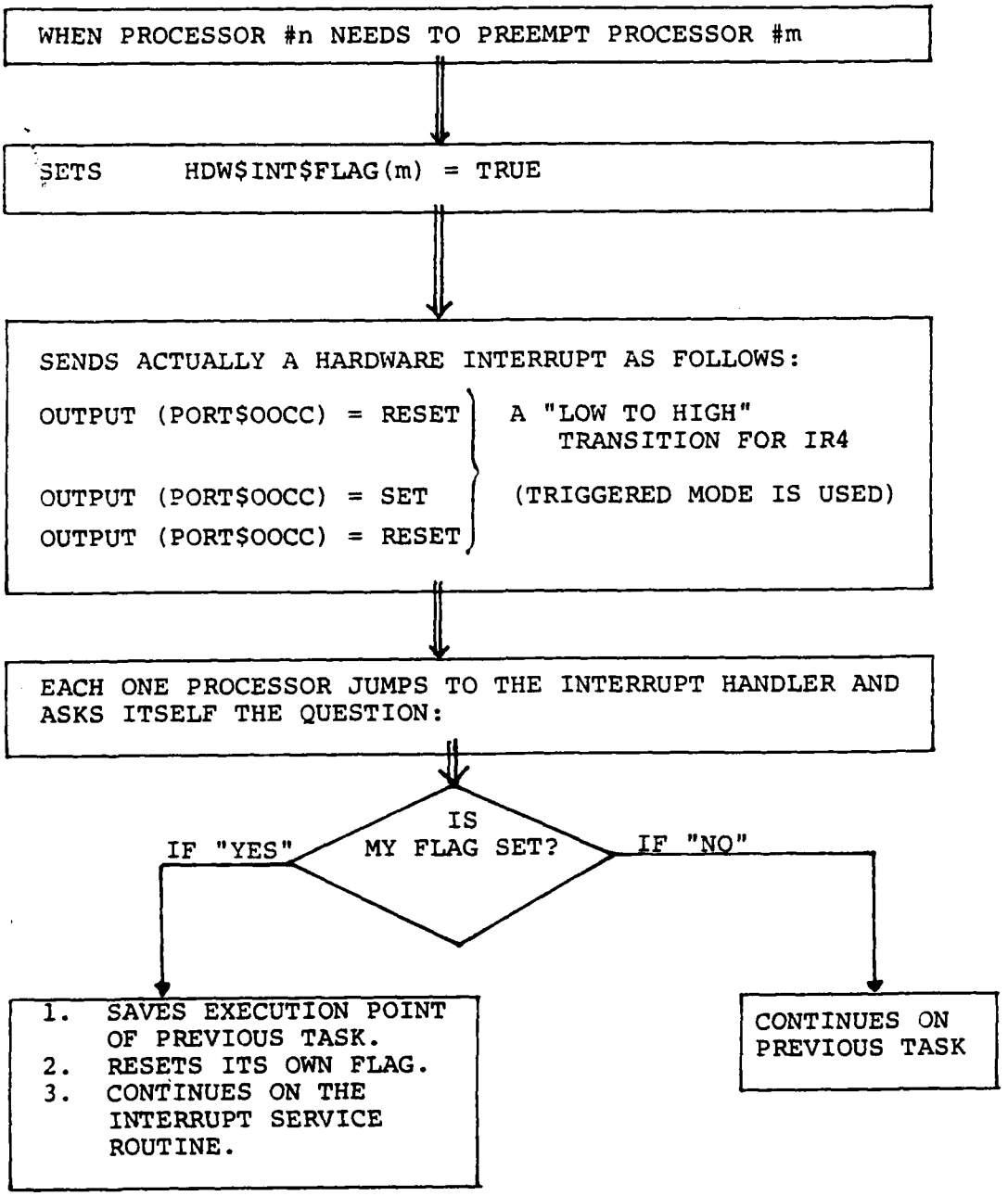


FIGURE 38. PREEMPTIVE HARDWARE INTERRUPT ALGORITHM

checks its own HDWINTFLAG. If the flag is not set, the processor continues on the previous task by using the IRET instruction. Otherwise, if the interrupt was destined for it, this processor saves the execution point of the previous task, resets its HDWINTFLAG and then continues on the interrupt service routine.

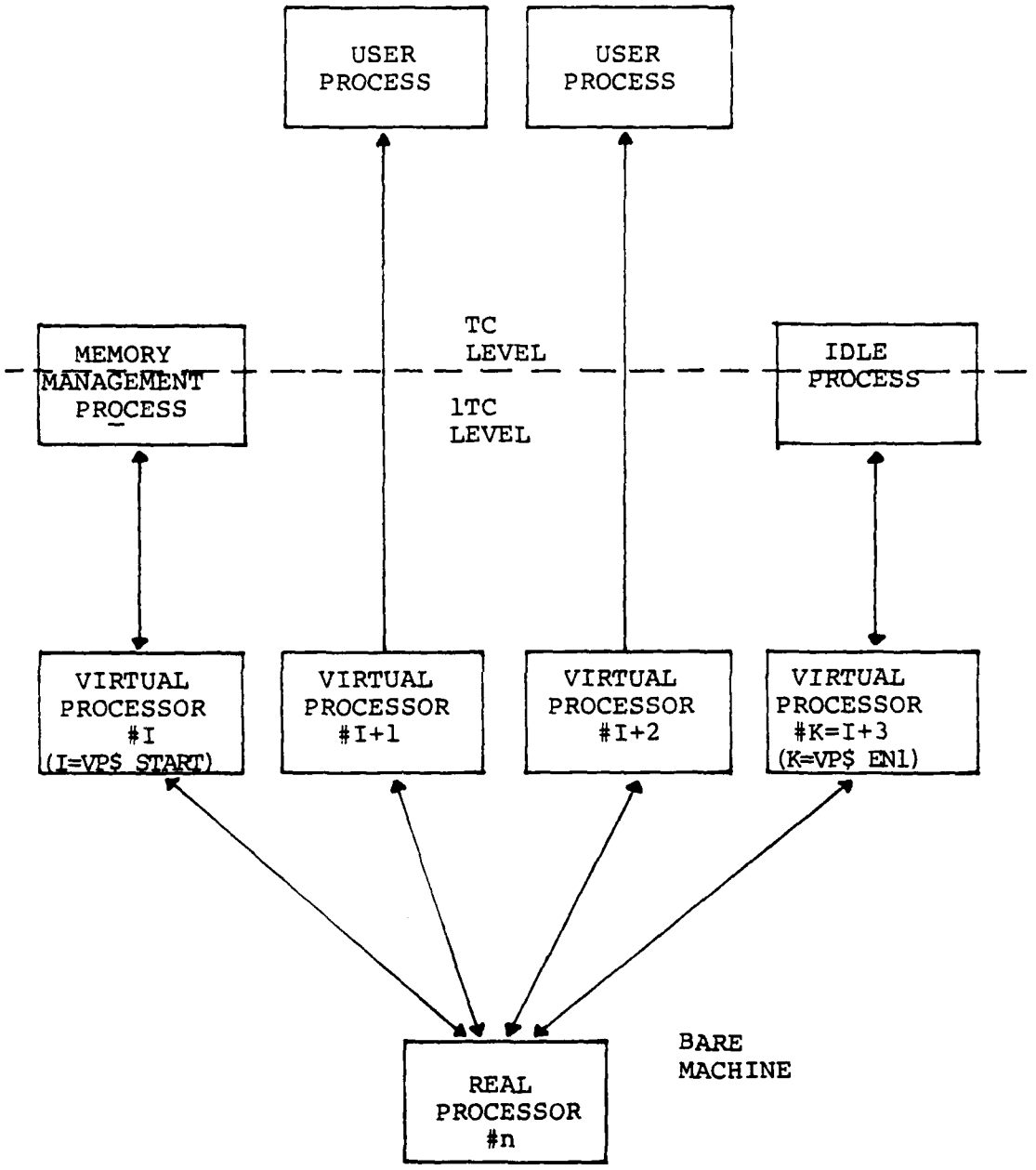
H. SYSTEM-WIDE DATABASES

The operating system is "database" or "control table" driven. There are several shared databases (shared segments) that reside in the global memory where any processor can access them to maintain and update the shared control data used by the operating system.

1. Virtual Processor Map (VPM)

The Inner Traffic Controller is the physical resource manager. The VPM is the principal global data base that maintains and updates the data used by the ITC to multiplex virtual processors among real processors and to create the extended instruction set that controls the virtual processor operation. The VPM is a system wide database and is kept in global memory (as a shared segment) to facilitate inter-virtual processor communication and synchronization.

Each physical processor has its own fixed set of virtual processors (four in the current implementation) used in multiplexing. See Figure 39. The first and fourth VP (VP$START and VP$END) are invisible to the TC level (invisible to the user processes) and are permanently bound to the memory



n = CPU$NUMBER = LOG$CPU$NUMBER

FIGURE 39. EACH REAL PROCESSOR POSSESSES FOUR VIRTUAL PROCESSORS

management process and idle process respectively. The VP$START has the highest priority (0 in this implementation) and the VP$END the lowest (255 or FFH). The remaining two have priority equal to the priority of the user processes bound to them. In this way the ITC recognizes that each real processor possesses four VP while the TC recognizes only two VP per real processor. A virtual processor mapping among the TC and ITC is needed to support this different VP view.

It is important to understand that this VP multiplexing among physical processors is an economic way for using the physical processor and physical resources in general. For example, by binding permanently the MMGT (Memory Management process) to a VP and assigning to this VP (VP$START) the highest priority, the MMGT process will occupy (run on) the physical processor each time there is reason (e.g., when some system event happens that requires a response by the MMGT). Otherwise another VP runs on this physical processor either the idle process or a user process. On the other hand if a real processor was permanently bound to the MMGT process, this physical resource would be idle whenever the MMGT process has nothing to do.

It is also important to note that the ITC executing on a physical processor is primarily concerned only with its set of the four VP. However, the performance of system-wide synchronization requires access to the remaining

virtual processors as well, so that signals may be used to alert other physical processors (we have discussed already the case of preemptive scheduling). This is accomplished by maintaining the Virtual Processor Map as a shared data base containing entries for all of the virtual processors in the system. Making it globally available facilitates communication between virtual processors on a system-wide scale. The Virtual Processor Map fields are shown in Figure 40.

The VPM INDEX starts from 0 to the value NR$RPS * VPS$PER$CPU-1, viz., number of real processors in the system multiplied by the number of virtual processors per real processor (four in current implementation) minus 1. This VPM INDEX represents a whole entry into VPM (a horizontal line in Figure 40). For example, VPM(0) represents the first entry (horizontal line), VPM(1) the second and so on.

The VP$ID field is used to support the VP mapping between the TC and ITC. Details will be discussed in paragraph I8 of this chapter..

The VP$STATE (virtual processor state) field reflects the present state of the virtual processor and can be any of "ready", "running", "waiting", or "idle". A ready virtual processor is bound to a process and is in "contention" for the physical processor. The running virtual processor is that virtual processor which is actually executing a process on this physical processor. The waiting state reflects physical resource management. The idle state is assumed by

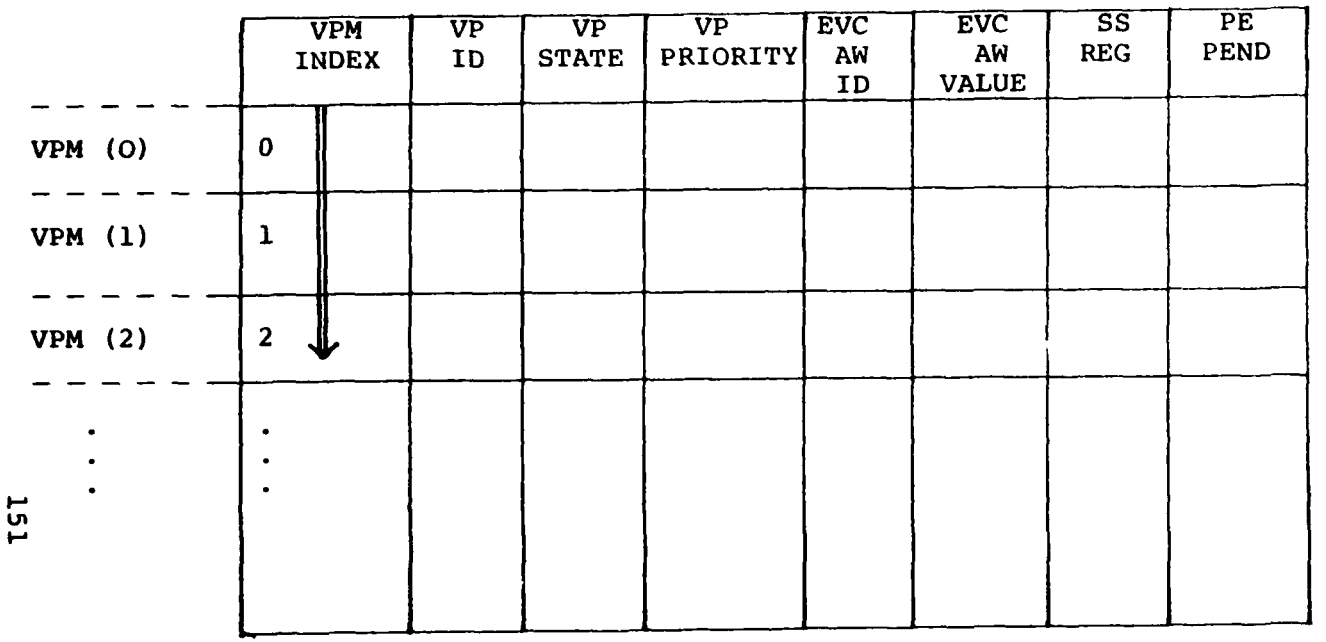


FIGURE 40. VPM (VIRTUAL PROCESSOR MAP)

a virtual processor which has no process bound to it. The idle state prevents the assignment of useless (idle) work to a physical processor. Figure 8 illustrates the state transitions made by the virtual processors. In paragraph C7b of Chapter II the possible transitions of state for a VP are described.

The VP$PRIORITY (virtual processor priority) field of the virtual processor is used in scheduling. The highest priority runnable virtual processor is selected to run. This priority is determined by the priority of the process bound to the virtual processor. The VP$START, which is permanently bound to the MMGT process has the highest priority (zero) and the VP$SEND the lowest priority (255 or FFH).

The EVCAWID (Awaited Eventcount Identifier) and EVCAWVALUE (Eventcount's waited value) fields are used in Inter-virtual processor communication and synchronization. Details will be discussed in the ITC$AWAIT and ITC$ADVANCE modules of the ITC later on in this chapter.

The SS$REG (Stack Segment Register Value) field defines the address space of the process bound to this VP. It holds the "process address space descriptor" (analogous to DBR in MULTICS). The execution point of the process is stored on the stack when the process is not actually running. This SS$REG is the only value which is required to access the address space of the process, viz., it is changed to swap processes.

The PE$PEND (Preempt Pending Flag) field is used for preemptive scheduling. It serves to transform a hardware interrupt sent to the physical process into a virtual preempt interrupt.

2. Active Process Table (APT)

The Traffic Controller multiplexes user (or application) processes among virtual processors. In this way the TC is responsible to manage the execution of user processes ("processes management"). It is noted, one more time, that since the processes are assigned to virtual processors (and not real processors), there is no effect on the user when real processors are added or deleted in the system, except, of course, for the change in performance. Most of the design and implementation, presented to the user, are independent of the physical configuration of the system.

The Traffic Controller's principal global data base is the Active Process Table (APT), shown in Figure 41. The entry for each process in the Active Process Table contains sufficient information about the process to enable a virtual processor to be bound to and execute it.

The APT INDEX starts from zero and grows as far as processes are loaded in the system. For example, the APT(0) represents the first entry (horizontal line) in the APT, APT(1) the second and so on.

The STATE field represents the state of a process and it can be either "ready", "running", or "blocked". A ready

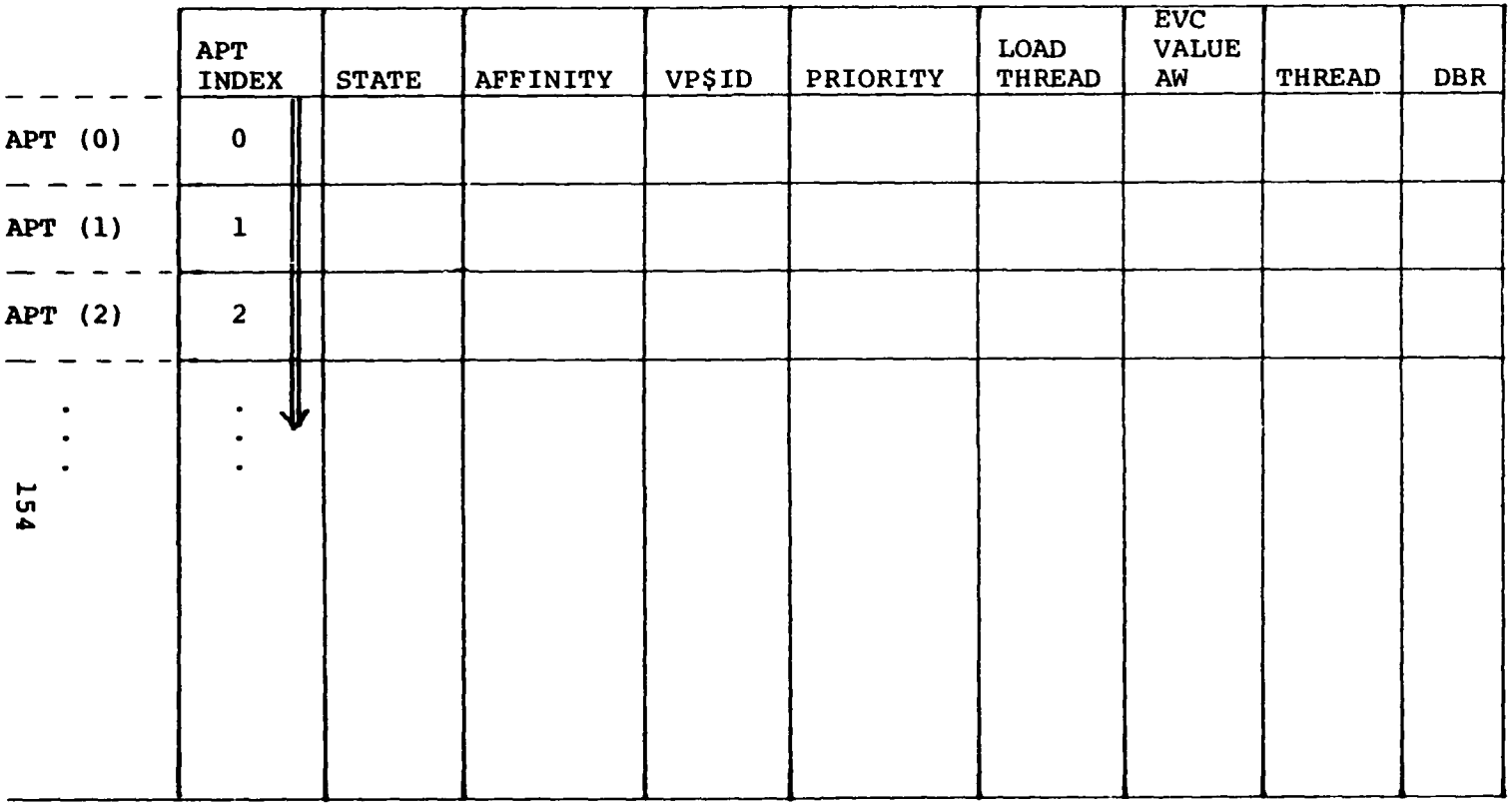


FIGURE 41. APT (ACTIVE PROCESS TABLE)

process is one which is not yet bound to a virtual processor but is ready to do so (it is in "contention" for VP). A running process is one which is bound to a virtual processor and, as far as the process is concerned, executing. The blocked state reflects inter-process synchronization. A process enters the blocked state when it realizes that it can no longer proceed and wishes to "give up" its virtual processor to wait until another process awakens it. This is important for the economic advantage of virtual processor multiplexing algorithm, viz., a process which can no longer run, waiting for the occurrence of an event frees the virtual processor which was bound to this process. The possible states of a process and the transitions among them are shown in Figure 7 and explained in paragraph C7a of Chapter II.

The AFFINITY field specifies the physical processor on which the process is currently loaded. It is possible to change this field during system "reconfiguration", Anderson [19].

The VP$ID (Identity Of Bound Virtual Processor) field serves to identify the virtual processor, if any, that the process is currently bound to. It is noted that the user processes are multiplexed among the two central virtual processors of each real processor as shown in Figure 39. The VP with identification number VP$START and VP$END are invisible to the TC and the user. The necessary mapping among VP$ID of the ITC and TC will be discussed in the ITCRETVPTC module in paragraph I8 in this Chapter.

The PRIORITY field specifies the priority of the process. In this system, priorities range in value from 0 to 255, with a priority of 0 being the highest. When a process is bound to a VP, the VP$PRIORITY field of the VPM corresponding to this specific VP, becomes equal to the PRIORITY field of the process.

The LOAD$THREAD (Loaded List Thread) field serves to implement the "Loaded List" of the ready, running and blocked processes. It contains a pointer to the next process in the Active Process Table which is loaded on the same microcomputer as this process. The meaning of this statement is that the "loaded list", which is a "linked list", is kept updated "per physical processor". A loaded process has its address space in primary storage; therefore it may be scheduled to run on a VP. In general, a process can be loaded on only a single physical processor at a time, due to the use of processor-local memory. The loaded list is ordered (sorted) by the priorities of the processes. Thus this field contains either a pointer to a process whose priority is less than or equal to that of this process or a nil pointer (viz., the last process on this Loaded List).

The EVC$VALUE$AW (Value of Eventcount Awaited) field reflects the event for which the process has blocked itself. It contains the value that the process is waiting for the eventcount to reach. When this specific eventcount reaches this value the process will awaken and its state will change from "blocked" to "ready". The usefulness of this field will

be better understood when describing the TC$AWAIT and TC$ADVANCE modules.

The THREAD (Block List Thread) field is used to implement the Blocked List. This is a "per eventcount" linked list of processes which are waiting on the same eventcount.

The DBR (Address Space Descriptor) field contains the process' address space descriptor. This is the identity of the process' stack which contains execution point information. The value used here is the base location in memory of the stack segment, viz., the Stack Segment (SS) Register value. This field is implemented exactly the same way as the SS$REG field of the VPM.

Above we described that the LOAD$THREAD field is used to implement a "per physical processor" linked list (the "load list") of the ready, running, and blocked processes and also that the THREAD field is used to implement a "per eventcount" linked list of the blocked processes waiting this eventcount. For better understanding of these statements we shall use an example later on, in paragraph H6.

3. Eventcount Table (EVC$TABLE)

The Eventcount Table is also a global data base for the TC level, as shown in Figure 42 and is used by the inter-process synchronization mechanism.

The EVC$TABLE INDEX starts from zero and grows as new events are added in the system by calls from the

application processes. For example, the EVC$TABLE(0) represents the first entry (horizontal line) in the EVC$TABLE, EVC$TABLE(1) the second and so on.

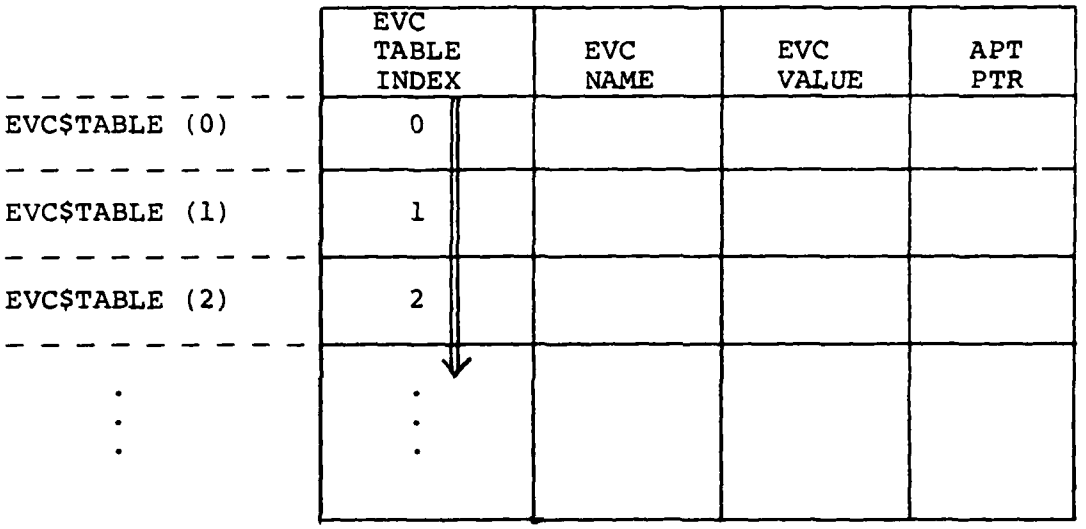


FIGURE 42. EVC$TABLE (EVENTCOUNT TABLE)

The EVC$NAME (eventcount name) field is a character array of six letters. The first five letters is the name given to the specific event by the user and the last letter is a delimiter (% is used). This name is used as the input argument of the TC$AWAIT and TC$ADVANCE operations.

The EVC$VALUE (Eventcount value) field holds the current value of the eventcount. Each time a TC$ADVANCE operation is executed, this value is incremented by one. Each time the TC$AWAIT or TC$ADVANCE is invoked, a comparison is made between this Eventcount current value and the awaited value to

decide if the state of the process will remain blocked or will be changed to ready.

The APT$PTR (Active Process Table Pointer) field is a pointer which points to the first member of the blocked list (the "per eventcount" link list discussed in previous paragraph) corresponding to this specific eventcount. The usefulness of this pointer will be better understood in the example promised in previous paragraph.

This structure also uses the variable EVENTS with initial value zero. The value of EVENTS is incremented by one each time the TC$CREATE$EVC (Traffic Controller Create Eventcount) is invoked by an application process. In this way the operating system keeps track how many events are currently in use for inter-process synchronization and communication.

4. Inner Traffic Controller Eventcount Table (ITCEVCTBL)

This is a global data base for the ITC level shown in Figure 43 and is used by the inter-virtual processor synchronization mechanism.

This table is a parallel structure with the previously described EVC$TABLE. The differences are: the EVC$NAME in this table is not a character array but just a number (0 to FFH). The reason is that this structure is invisible for the user and therefore it is not necessary to spend execution time to improve the "user interface" (viz., takes more time when we search the EVC$TABLE to find an eventcount name consisted of six characters).

	ITC EVC TBL INDEX	EVC NAME	EVC VALUE
ITCEVCTBL (0)	0		
ITCEVCTBL (1)	1		
ITCEVCTBL (2)	2		
.	.		
.	.		
.	.		

TABLE 43. ITCEVCTBL (INNER TRAFFIC CONTROLLER EVENTCOUNT TABLE)

This structure also uses the variable ITC$EVENTS (Inner Traffic Controller Events) to keep track of how many events are currently in use in the ITC level, for inter-virtual processor communication and synchronization.

5. System Configuration Data Segment (SCDS)

This is also a shared (global) segment containing the following information:

NR$RPS (Number of Real Processors) provides the information how many physical processors are currently used in the system.

NR$VPS (Number of Virtual Processors) provides the number of virtual processors used in the system. It is

noted that NR$VPS = NR$RPS * VPSPERCPU, e.g., the number of virtual processors always equals to the number of real processors multiplied by the number of virtual processors per real processor, that is 4 in the current implementation.

The array HDWINTFLAG(n) (Hardware Interrupt Flag), is used by the hardware interrupt mechanism for directing an interrupt to a specific physical processor. Initially all the members of this array are set to zero. The number of these members is equal to NR$RPS (n = NR$RPS - 1). There is one-to-one mapping among HDWINTFLAG and CPU$NUMBER (or LOG$CPU$NUMBER), e.g., HDW$INT$FLAG(m) corresponds to CPU$NUMBER = m. The usefulness of these flags has already been discussed in paragraph G of this chapter.

The array LOAD$LIST(n) (Load List), is used in the implementation of the linked list of the processes loaded to each physical processor (The "Load List" discussed in the previous paragraph 2, above APT). Initially all the members of this array are set to zero. The number of these members is again equal to NR$RPS (n = NR$RPS - 1). There is also one-to-one mapping among LOAD$LIST and CPU$NUMBER. LOAD$LIST(m) points to the currently highest priority process (independent of whether this process is ready, running, or blocked) loaded on the physical processor with CPU$NUMBER (or LOG$CPU$NUMBER) = m.

6. An Example for Loaded Lists and Blocked Lists

It is now feasible to present an example to illustrate the interactions among LOAD$THREAD, THREAD, APT$PTR, and LOAD$LIST.

It is noted that it is important for the reader to understand the following example before proceeding into the details of the following paragraphs I (about the Inner Traffic Controller) and especially K (about the Traffic Controller).

Figure 44 illustrates the interactions for this example. The APT, SCDS, and EVC$TABLE tables of Figure 44 do not show all their members but only the ones needed to demonstrate the ideas. It is supposed that 11 processes corresponding to APT(0) through APT(10) entries of the APT have been loaded on three different physical processors with AFFINITY (CPU$NUMBER or LOG$CPU$NUMBER) 0, 1 and 2.

Three linked "Loaded lists" are generated by the operating system, one "per physical processor". These three linked lists are sorted (ordered) by the priorities of the loaded processes. For example, the LOAD$LIST(1) of the SCDS, corresponding to the physical processor with AFFINITY = 1 (LOGCPUNUMBER = 1) points to the highest priority process loaded on physical processor #1. It is shown in the Figure 44, that LOAD$LIST(1) = 2. The meaning is that the LOAD$LIST(1) (the header of this linked list) points to the entry 2 of the APT (APT(2)). In entry 2 of the APT, there

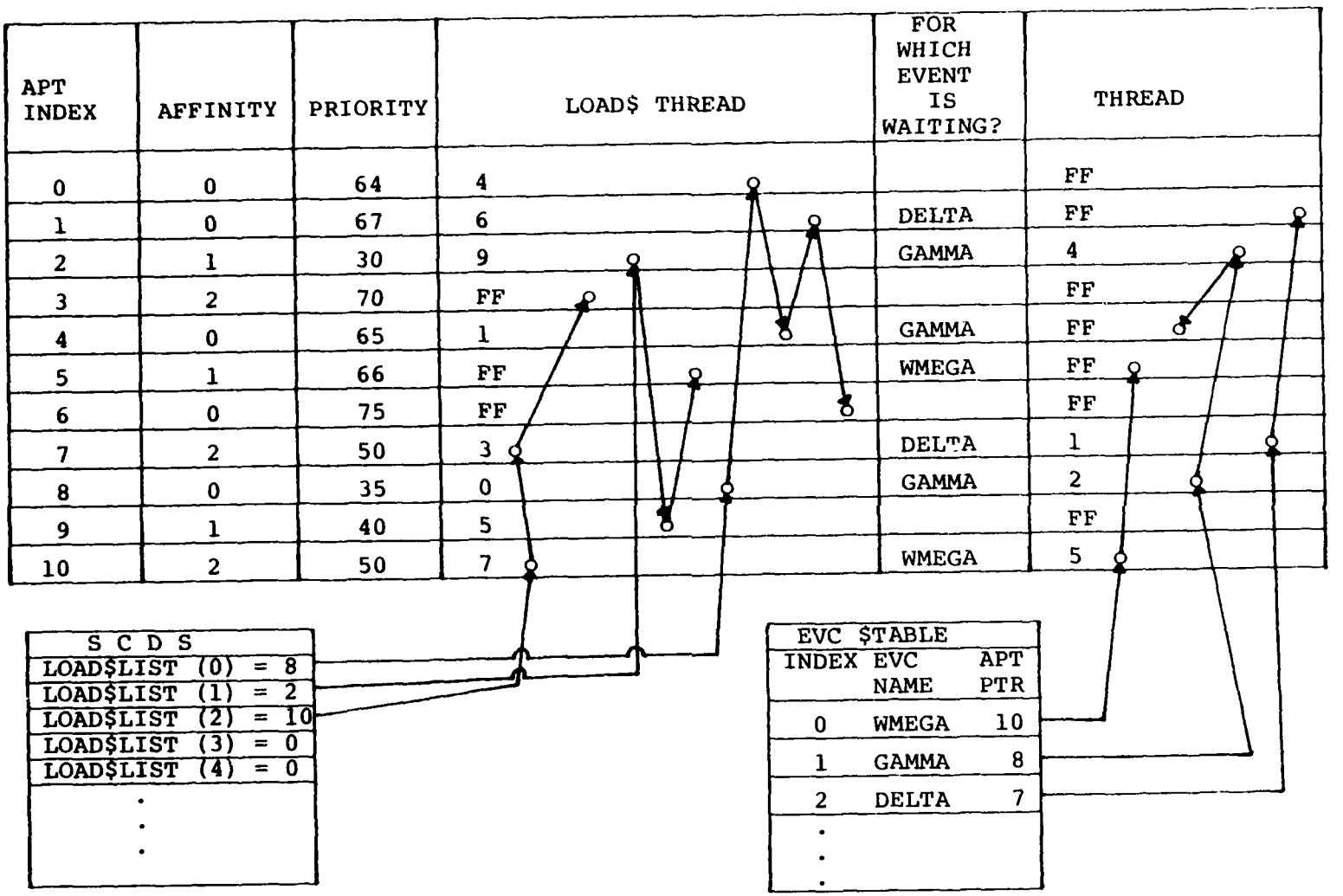


FIGURE 44. LOADED LISTS AND BLOCKED LISTS

is loaded a process on physical processor #1 (AFFINITY = 1) and its priority is 30. On the same processor are loaded two more processes corresponding to the entries 5 and 9 of the APT but their priorities are lower (66 and 40 respectively).

The LOAD$THREAD corresponding to APT(2) is equal to 9. The meaning is that the next process loaded on this physical processor #1 is in the entry 9 of the APT. Indeed the AFFINITY of APT(9) is also equal 1, and its LOAD$THREAD field is equal to 5. The meaning is that the next loaded process on this physical processor is in entry 5 of the APT. The LOAD$THREAD of APT(5) is equal to FF (the NIL pointer). This means this is the last process (the lowest priority process) loaded on physical processor #1. To summarize, we have LOAD$LIST(1) = 2 pointing to APT(2) which is the highest priority process (with priority 30) loaded on this physical processor. This process points to the entry 9 (it has priority 40) and this second process in turn points to the entry 5 which contains the third process (with priority 66) and its LOAD$THREAD = FF meaning it is the last one in this linked list.

Similarly, it is possible now to easily follow the path of the remaining two loaded lists (the linked lists of the processes loaded on physical processors #0 and #2).

It is also supposed that several of these processes are in the blocked state waiting the occurrence of some

event. There exist three events in the EVC$TABLE with names WMEGA, GAMMA, and DELTA. The processes corresponding to the APT entries 5 and 10 are waiting for the occurrence of the event WMEGA, the processes corresponding to the APT entries 2, 4, and 8 are waiting for the occurrence of the event GAMMA and finally the processes corresponding to the APT entries 1 and 7 are waiting for the occurrence of the event DELTA.

Three linked "Blocked lists" are generated by the operating system one "per eventcount". For example, the APT$PTR corresponding to the EVC$NAME WMEGA is equal to 10. The meaning is that the EVC$TABLE(0).APT$PTR points to the entry 10 of the APT and indeed this process is waiting the occurrence of the event WMEGA. The THREAD field of the APT(10) is equal to 5. The meaning is that the process in APT(5) is also waiting the occurrence of the same event, and finally the THREAD field of APT(5) is equal FF meaning that there is no other process waiting the occurrence of the event WMEGA. It is noted that these linked lists are per eventcount and they link processes waiting the specific event independent of the processor on which they are loaded.

Similarly it is possible now to follow easily the path of the remaining two blocked lists corresponding to the events GAMMA and DELTA.

7. Locks Table (LOCKS)

This small global table consists only of the two following bytes: APT$LOCK and VPM$LOCK (Active Process Table Lock and Virtual Processor Map Lock). These two locks are used to prevent race conditions when accessing the shared data bases APT and VPM. The meaning and usefulness of these locks have already been discussed.

8. Processor Data Segments (PRDS)

This segment doesn't contain system-wide (*global*) data but "local" data, viz., data used for the specific microcomputer on which this segment is loaded. There exist a PRDS "per physical processor". This segment contains only the structure shown in Figure 45.

```
DECLARE PRDS STRUCTURE
(CPU$NUMBER                BYTE,
 VP$START                  BYTE,
 VP$END                    BYTE,
 VPS$PER$CPU               BYTE,
 IDLE$DBR                  WORD,
 COUNTER                   WORD,
 VIRT$INT$VECTOR           POINTER,
 HDW$INT$VECTOR            POINTER)
```

FIGURE 45. PROCESSOR DATA SEGMENT STRUCTURE (PRDS STRUCTURE)

The CPU$NUMBER (A "unique" identification number for the specific physical processor) field, is assigned to each physical processor during system initialization and is equal to the LOGCPUNUMBER (Logical CPU number) passed as input argument to the module ITC$INIT (Inner Traffic Controller Initialization) which will be discussed in paragraph Ib of this chapter. Anderson [19] describes in his thesis the details about system initialization.

The VP$START and VP$END fields define the identification number of the first and last virtual processor assigned to the specific physical processor. For example, in this implementation, the physical processor with identification number CPU$NUMBER = 0 corresponds to VP$START = 0 and VP$END = 3, the physical processor with CPU$NUMBER = 1 corresponds to VP$START = 4 and VP$END = 7, and so on.

The VPSPERCPU (Virtual processors per CPU) field, determines the number of virtual processors assigned to each physical processor. In the current implementation this number is fixed and equal to 4.

The IDLE$DBR (Address space descriptor for the idle process) field determines the address of the base of the Idle Stack (IDLE$STACK) which is used by the Idle Process. Details about this stack will be discussed in the ITC$INIT module in paragraph Ib of this chapter.

The COUNTER field is a software counter. By containing this member in the PRDS structure, which is local to each

microcomputer an array of software counters is automatically generated with one-to-one correspondance to the physical processors. These counters are initialized to zero, and will be used to monitor the system's performance and the effectiveness of the partitioning of the application programs. Details will be discussed in paragraph Jb of this chapter. VIRTINTVECTOR and HDWINTVECTOR (Virtual interrupt vector and hardware interrupt vector) fields determine the address where the CPU of the specific microcomputer has to transfer the program control when it receives a virtual or a hardware interrupt. When a CPU receives a virtual interrupt, it transfers program control to the Traffic Controller Preemption Handler (TCPEHANDLER). This module will be described in the paragraph K, later on in this chapter. When a CPU receives a hardware interrupt, it transfers the program control to the hardware interrupt handler of the Inner Traffic Controller Scheduler (VPSCHEDULER). This module will be discussed in the paragraph Ia, later on, in this chapter.

9. Sequencer Table (SEQ$TABLE)

This is a global data base for the TC level shown in Figure 46 and is used by the inter-process synchronization mechanism.

The SEQ$TABLE INDEX starts from zero and grows as new sequencers are added to the system by the application processes. For example, SEQ$TABLE(0) represents the first

entry (horizontal line) in the SEQ$TABLE, SEQ$TABLE(1) the second and so on.

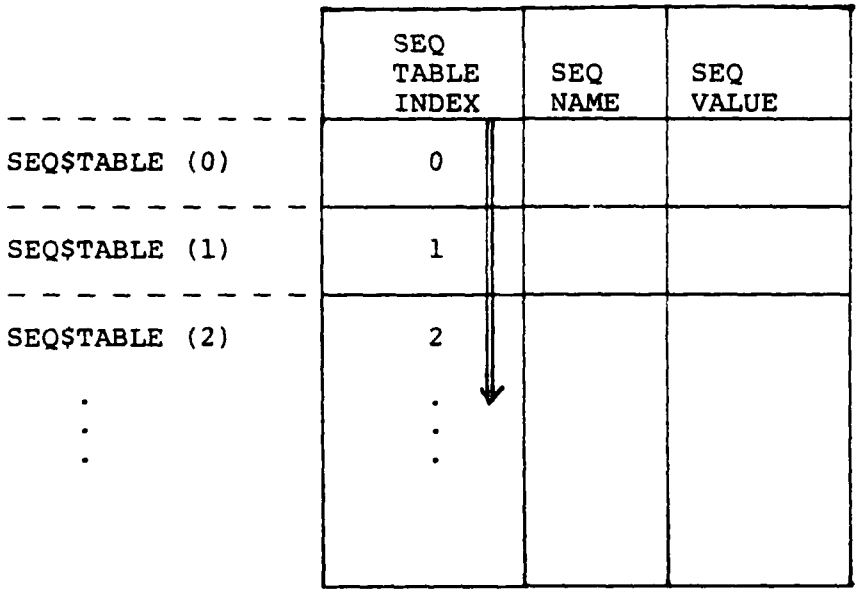


FIGURE 46. SEQ$TABLE (SEQUENCER TABLE)

The SEQ$NAME (Sequencer name) field is a character array of six letters. The first five letters is the name given to the specific sequencer by the user and the last letter is a delimiter (% is used). This name is used as the input argument of the TC$TICKET (Traffic Controller TICKET) operation.

The SEQ$VALUE (Sequencer value) field holds the current value of the sequencer. Each time a TC$TICKET operation is executed on the specific sequencer this value is incremented by one.

This structure also uses the variable SEQUENCERS with an initial value of zero. The value of SEQUENCERS is incremented by one each time the TC$CREATE$SEQ (Traffic Controller Create Sequencer) is invoked by an application process. In this way the operating system keeps track of how many sequencers are currently in use for inter-process communication and synchronization.

I. THE INNER TRAFFIC CONTROLLER

The Inner Traffic Controller comprises the lower level of processor multiplexing (Level 1 of this virtual machine). It multiplexes physical processors among a fixed set (four in the current implementation) of virtual processors. It provides inter-virtual processor communication and synchronization, supports the management of physical resources and manages the system's interrupt structure.

The Inner Traffic Controller creates a set of four virtual processors with the following extended instruction set: ITC$AWAIT, ITC$ADVANCE, ITC$LOAD$VP, IDLEVP, ITCSEND$PREEMPT, and ITC$RET$VP. It also contains the internal routines HARDWARE$INT, LOCKVPM, UNLOCKVPM, CHECK$PREEMPT, RDYTHISVP and SWAPDBR.

ITC$AWAIT and ITC$ADVANCE (Inner Traffic Controller AWAIT and ADVANCE) provide an inter-virtual processor synchronization mechanism used within the kernel to provide multiprogramming. This multiprogramming is realized by invoking the scheduling procedure GETWORK, of the ITC, which multiplexes these four

virtual processors on a physical processor. Which VP will finally run on the physical processor is decided by the VPSCHEDULER (Inner Traffic Controller Scheduler).

ITC$LOAD$VP (Inner Traffic Controller Load Virtual Processor) performs the "binding" of a new process to a virtual processor. It is called by the TC$SCHEDULER (Traffic Controller Scheduler) when a process has been selected for the VP.

IDLE$VP (Idle this VP) is the ITC$LOAD$VP's counterpart. It is called by the TC$SCHEDULER in case that there exist no runnable process for the VP. The virtual processor will be idled (enter the "idle state").

CHECK$PREEMPT and ITC$SEND$PREEMPT (Check for Pending Preempt Interrupt and ITC Send Preempt Interrupt) create a virtual processor interrupt mechanism. CHECK$PREEMPT, when it is invoked within the ITC, checks the PE$PEND (Preemption Pending Flag) field of the VPM to determine if it is set or reset for the specific VP. ITC$SEND$PREEMPT is invoked from level 2 (TC$ADVANCE) when the Traffic Controller desires to load a new process on a virtual processor that is not scheduled.

ITCRETVP (Inner Traffic Controller Return Virtual Processor's identification number), when it is invoked, provides the information which VP is currently scheduled (running) on the physical processor. This identity is only valid so long as the APT is locked. The identity of a particular VP must be known in the virtual environment, just as the identity of

a physical processor is required to be known in the multi-processor system.

HARDWARE$INT (hardware interrupt) is used within the ITC to send a hardware interrupt from one physical processor to another. The purpose is to support preemptive scheduling needed in the real-time processing.

LOCKVPM and UNLOCKVPM (Lock and Unlock the Virtual processor map) are used to set or reset a software lock on the shared (global) VPM data base to assure there are no race conditions.

RDYTHISVP (Ready this VP) is used to change the state of the currently "running" VP to "ready".

SWAPDBR (Swap DBR) is a function within the Inner Traffic Controller Scheduler and is used to change the address space when a new process is scheduled to run when the previous process has been completed or blocked.

The details of the Inner Traffic Controller modules will be discussed below:

1. Virtual Processor Scheduler (VPSCHEDULER)

This module is responsible for making the scheduling decisions for virtual processors. It selects the highest priority virtual processor from the set of four virtual processors assigned to the physical processor and schedules it. There are two distinct entry points to the VPSCHEDULER, the normal entry and the interrupt entry.

The normal entry point is used by other Inner Traffic Controller modules to activate VPSCHEDULER when a virtual processor gives up the physical processor on its own. The preempt interrupt entry point is used in response to a hardware preempt interrupt from another physical processor.

VPSCHEDULER next searches through the fixed set of virtual processors for the highest priority "eligible" virtual processor. In this implementation the definition of eligible includes not only a ready VP but also the combination of an idle state and a pending virtual preempt interrupt. This allows an idle virtual processor to run so that it may field the interrupt and bind itself to a new process. The idle process that was bound to the virtual processor was essentially useless up to this point. It now provides an address space in which the virtual processor can execute when binding to a new process.

Having selected some eligible virtual processor, the VPSCHEDULER proceeds to bind the selected virtual processor to the physical processor. It does so by unbinding the currently running virtual processor. In doing so, the Stack Pointer Register (SP) value, and the Base Pointer Register (BP) value are saved in known locations on the process' stack. The process' execution state (point) had already been saved.

Binding the selected virtual processor is begun by changing the Stack Segment (SS) Register value to that of the selected virtual processor. Once this change has been made, execution has actually swapped to the new process address space. Binding is completed by retrieving the previously saved stack Pointer Register value and the Base Pointer Register value from the newly acquired stack.

The last step is to actually return to the proper place in the VPSCHEDULER. If a preempt interrupt invoked VPSCHEDULER, an interrupt return will be executed and CHECKPREEMPT will see if a virtual preempt interrupt is pending. If a preempt interrupt is found to be pending, the program control will be transferred to the location specified by PRDS.VIRTINTVECTOR (viz., to the Traffic Controller's preempt handler).

There is one other internal module for the Virtual Processor Scheduler, the hardware interrupt handler. It is used to handle hardware preempt interrupts. The program control is transferred in this module each time the HARD-WARE$INT module of the ITC is invoked. Details about the hardware preempt interrupt mechanism have already been discussed in the paragraph G of this Chapter. (For the algorithm see Figure 38).

2. ITC$INIT (Inner Traffic Controller Initialization)

This module together with the following KERNEL$INIT perform part of the system initialization by initializing the

stack of the Idle Process and also the stack of the Memory Management Process. These two system processes run conceptually between the TC and ITC levels as shown in Figures 5 and 9. These two processes are scheduled by the VPSCHEDULER (the ITC scheduler), and are "invisible" to the TC$SCHEDULER (generally to the TC level). That means there is no entry into the APT (Active Process Table) for these two processes. Also the stack initialization for these processes is different from the corresponding initialization of an application process stack. Details about these two system processes will be discussed after the completion of the ITC level.

This module just calls the KERNEL$INIT module and then calls the VPSCHEDULER that schedules the highest priority virtual processor (VP #0) to run. VP #0 is permanently bound to the Memory Management Process.

ITC$INIT accepts two input arguments, CPU$NUMBER (that is equal to the LOGCPUNUMBER, logical CPU number) and PHYSCPUNUMBER (physical CPU number). These two arguments LOGCPUNUMBER and PHYSCPUNUMBER are given values during the system initialization [19].

ITC$INIT is the entry point for the distributed operating system.

3. KERNEL$INIT (Kernel Initialization)

This module is called only by the ITC$INIT and is executed by each processor once during the system initialization. It declares the IDLE$STACK and MGMT$STACK

(Idle Process Stack and Memory Management Stack respectively) as based structures. It then initializes these two stacks by initializing the header of the stack and the register's array and then initializing the maximum stack length, and the process' initial code segment (CS) register, instruction pointer (IP) register and the flags. (See Figure 22).

Then the program control returns into ITC$INIT module.

4. GET$COUNTER (Get Current Value of COUNTER)

This is just a "utility function" called only by the Idle Process. It gets the current value of the counter (which is a member of the PRDS, (see Figure 45)) and returns that value to the Idle Process.

5. UPDATE$COUNTER (Update the Value of COUNTER)

This is also a "utility function" called only by the Idle Process and has the purpose to update (increment by one) the current value of the COUNTER. The usefulness of these two utility functions will be discussed when describing the Idle Process.

6. GET$CURRENT$DBR (Get Current DBR)

This is also a "utility function" and is called only by the VPSCHEDULER. When making an implicit call to the ITCRETVP (discussed below), it finds the identity (VP number) of the currently running virtual processor and then finds and returns the content of the Stack Segment (SS) register, corresponding to the specific running VP. Recall

that the SS register is used in this design in a manner analogous to the DBR in the MULTICS system. This DBR value is used by the VPSCHEDULER to identify the right address space and continue execution after receiving a hardware interrupt.

We note here that each time a module returns a function value, this value in PL/M-86 always goes into the accumulator (AX) register.

7. ITCRETVP (Inner Traffic Controller Return VP Number)

This is also a "utility function" used by the Inner Traffic Controller and Traffic Controller modules. ITCRETVP searches the Virtual Processor Map and determines the identity of the virtual processor that is currently running on the physical processor. It simply checks for the virtual processor among the virtual processors assigned to the physical processor which is in the running state. ITCRETVP then returns its result as a function value into the AX (accumulator) register. It will return either the identity of the virtual processor (the virtual processor's index in the Virtual Processor Map) or a "not found" error code.

8. ITCRETVPTC (ITC Return VP number for TC)

It is a "utility function" which is used to perform the VP mapping between the TC and ITC levels as already mentioned in paragraphs H1 and H2 (about VP$ID Field) of this Chapter.

All the four VP's in the Figure 39 are visible to the ITC. The two central VP's are visible to the TC while VP$START and VP$END are invisible. The user processes are multiplexed among these two central VP's of each physical processor.

The ITCRETVPTC when called by the TC, it calls in turn the ITCRETVP to obtain the currently running VP (its index in VPM). It then performs the mapping shown in Figure 47, and finally returns the corresponding VP identification number for the TC (VP$ID in Figure 47).

9. ITC$LOAD$VP (Inner Traffic Controller Load Virtual Processor)

This module performs the "binding" of a new process to a virtual processor. It is called by the Traffic Controller Scheduler (TC$SCHEDULER) when a process has been selected for the virtual processor. LOAD$VP requires two input parameters, the priority of the new process and the address space descriptor (the Stack Segment Register value). It then swaps in the new process onto the virtual processor which is currently running. ITC$LOAD$VP only operates on the virtual processor which is running on the physical processor.

Binding is accomplished by updating the Virtual Processor Map. The Inner Traffic Controller utility function ITCRETVP is used to obtain the identity of the running virtual processor. When complete, the virtual processor will have a new priority and process address space descriptor

VPM INDEX VPM (n)	AFFINITY OR PRDS. CPU$NUMBER	VP$ID
0 (VP$START FOR RP #0)	0	FF
1		0
2		1
3 (VP$END FOR RP #0)		FF
4 (VP$START FOR RP #1)	1	FF
5		2
6		3
7 (VP$END FOR RP #1)		FF
8 (VP$START FOR RP #2)	2	FF
9		4
10		5
11 (VP$END FOR RP #2)		FF
. . .	. . .	. . .

FF = INVISIBLE FOR TC

RP = REAL PROCESSOR

MAPPING:

VP$ID = (VPM INDEX) - (PRDS.CPU$NUMBER * 2 + 1)

FIGURE 47. VIRTUAL PROCESSOR MAPPING BETWEEN ITC AND TC

(corresponding to the priority and address space of the process just bound to it). ITC$LOAD$VP completes by calling VPSCHEDULER to reschedule the virtual processor.

10. IDLE$VP (Idle this Virtual Processor)

This function is ITC$LOAD$VP's counterpart. It is called by the TC$SCHEDULER (Traffic Controller Scheduler) in the event that a runnable process is not found for the virtual processor. In this case the virtual processor will be idled (enter the idle state) and the Idle Process will be bound to it. In the Virtual Processor Map, the virtual processor's state will be marked as idle, the address space descriptor for the Idle Process will be entered in the Address Space of Bound Process field. The idle state ensures that the idle process is not actually run by taking the virtual processor entirely out of contention for the physical processor, with which this virtual processor is associated.

At some later point, the virtual processor may be placed back in "contention" for resources. This will occur when the virtual processor is "preempted". With the combination of an "idle state" and a "pending preempt", the virtual processor is treated the same way as a "ready" virtual processor (We shall clarify that statement when describing the GETWORK module). This allows the virtual processor to keep busy by expediting its binding to a process.

Lastly IDLE$VP calls VP$SCHEDULER in order to "give up" the physical processor.

11. CHECK$PREEMPT (Check for Pending Preempt Interrupt)

This module is called by the VPSCHEDULER during the execution of a "virtual interrupt return". It checks for a pending preempt interrupt meant for the virtual processor, which has been selected to run (the running VP) by the VPSCHEDULER. To accomplish this it checks the virtual processor's "preempt pending flag" (PE$PEND) in the VPM (Virtual Processor Map). If the preempt pending flag is set, the CHECK$PREEMPT will reset it and return the found value (flag "on" or "down") to the VPSCHEDULER. In this way the VPSCHEDULER is informed about the state of PE$PEND flag and it will use this information to decide which VP will run (see GETWORK module below).

12. GETWORK

It is a function call. Initially it sets its local variable PRI (Priority) equal to the lowest possible priority. (In this implementation, the lowest priority is 255 and the highest is 0) and SELECTED$DBR (selected address space) equal to IDLE$DBR (the address space for the idle process).

It then searches the VPM (Virtual Processor Map) to find the highest priority, "eligible" to run, virtual processor. In this implementation eligible to run for a virtual processor means it is either in the "ready" state, or the "idle" state with a "virtual preempt pending" (PE$PEND is set).

Using the above criterion, GETWORK selects an eligible processor, sets the SELECTED$DBR and PRI equal to the

corresponding VPM values SS$REG and VP$PRIORITY respectively for the selected VP, and then sets its state to "running" and finally returns the SELECTED$DBR into the Accumulator.

If after the above search no eligible VP is found, it defaults SELECTED$DBR = IDLE$DBR and the idle process will run.

13. ITC$SEND$PREEMPT (Inner Traffic Controller Send Preempt Interrupt)

This module is responsible for actually sending preempt interrupts. It is called by the Traffic Controller Advance module. ITC$SEND$PREEMPT requires two arguments, the identity of the virtual processor which is to be preempted and the identity of the physical processor to which that virtual processor is associated.

It first locks the VPM (Virtual Processor Map) and then sets the virtual processor's PE$PEND (Preempt Pending Flag). This is all that is done when the virtual processor to be preempted is associated to the physical processor, which is the transmitter (executing the ITC$SEND$PREEMPT module). In other words, when the TGT$CPU (the input argument showing the identity of the physical processor possessing the virtual processor for which the virtual preempt interrupt is destined) is equal to the CPU$NUMBER (the identity of the physical processor executing ITC$SEND$PREEMPT).

Otherwise, after setting the PE$PEND the ITC$SEND$-PREEMPT calls the HARDWARE$INT procedure (see next paragraph)

to generate a hardware interrupt for the physical processor possessing the virtual processor to be preempted.

Finally the ITC$SEND$PREEMPT unlocks the VPM and returns to the TC$ADVANCE (the module responsible for preemptive scheduling).

14. HARDWARE$INT (Hardware Interrupt)

This procedure requires as its input argument the CPU$-NUMBER, viz., the identity of the physical processor for which the hardware interrupt is destined. HARDWARE$INT procedure first sets the "global" hardware interrupt flag corresponding to this physical processor (HDWINTFLAG(CPU)). It then sends a hardware interrupt by outputting in the parallel PORT "C", first a "0" then an "80H" and again a "0". Finally the program control returns to the calling procedure. The details about this hardware preempt interrupt already have been discussed in paragraph G of this chapter. HARDWARE$INT is called only by the ITC$SEND$PREEMPT and ITC$ADVANCE modules.

15. LOCKVPM (Lock Virtual Processor Map)

This small module uses a built-in PL/M-86 procedure called LOCKSET which is an "indivisible test-and-set semaphore" to implement a software lock called LOCK$VPM in the VPM which is the central shared data base in the Inner Traffic Controller Level (see Figure 9). Because this global data base can be accessed (read and write capability)

by all the virtual processors, this lock is used to prevent "race conditions".

16. UNLOCKVPM (Unlock Virtual Processor Map)

This module is the counterpart of the above LOCKVPM. Each time we have to access the VPM, we first lock the VPM$LOCK. When the access task is finished, we have to unlock this VPM$LOCK, so that another virtual processor can access it.

17. RDYTHISVP (Ready this Virtual Processor)

This module first finds which Virtual Processor is currently running by calling implicitly ITCRETVP and then changes the state of this VP from "running" to "ready".

18. ITC$LOCATE$EVC (Inner Traffic Controller Locate Eventcount)

This is a utility function. It returns the index of an ITC Eventcount in the ITC Eventcount Table (ITCEVCTBL). It is called only by ITC$AWAIT and ITC$ADVANCE described below. The input argument is the name of this ITC Eventcount. ITC$-LOCATE$EVC attempts to match the name given to it with one in the ITCEVCTBL. If a match is found, it returns the index to the calling procedure in the AX (Accumulator) Register as a function value. Otherwise, it returns an error code.

19. ITC$AWAIT (Inner Traffic Controller AWAIT)

ITC$AWAIT is an inter-virtual processor synchronization primitive. It is "invisible" (not accessible) to the user processes and is used only by the operating system in

the management of physical resources. It allows a virtual processor to wait for the occurrence of an ITC Eventcount.

ITC$AWAIT expects two input arguments, the name of the Eventcount and the value of the event to be awaited.

Upon invokation ITC$AWAIT locks the VPM. It then finds first which Virtual Processor is running by making an implicit call to the ITCRETVP and then finds the index of the Eventcount in the ITCEVCTBL by making an implicit call to the ITC$LOCATE$EVC. It then compares the current value of the Eventcount, obtained from the ITCEVCTBL with the value passed in the call. If the current value of the Eventcount is found to be less than the value of the input argument, then the virtual processor will enter the "waiting" state and "gives up" the physical processor.

This change of the virtual processor's state from "running" to "waiting" will be reflected in the VPM. The input arguments will also be entered in the VPM in the EVCAWID (Identity of the Awaited Eventcount) and the EVCAWVALUE (Eventcount Awaited Value) fields.

Otherwise, if the current value of the Eventcount is found to be equal or greater than the value of the input argument, then the state of this virtual processor will be changed from "running" to "ready".

Finally, in both cases the virtual processor will give up the physical processor by calling the VPSCHEDULER,

which will bind another (or possibly the same[1]) virtual processor to this physical processor. Upon the return from the VPSCHEDULER, the VPM will be unlocked.

20. ITC$ADVANCE (Inner Traffic Controller ADVANCE)

ITC$ADVANCE is an inter-virtual processor synchronization primitive. It also is "invisible" to the user processes and is used only by the operating system in the management of the physical resources. It expects one input argument, the name of the ITC Eventcount to be advanced.

Upon invocation, the VPM is locked. ITC$ADVANCE then finds which VP is running by making an implicit call to the ITCRETVP to change the state of this VP from "running" to "ready". It then finds the index of the Eventcount in the ITCEVCTBL by making an implicit call to the ITC$LOCATE$EVC, and the eventcount's value in this table is incremented by one.

ITC$ADVANCE then compares this incremented value with the events waited for by the other virtual processors which are synchronizing on the same eventcount. All those virtual processors whose Eventcount Awaited Value field (EVCAWVALUE) in the VPM is less than or equal to the current value of the eventcount are set to the "ready" state. This is the "broadcast effect" discussed in paragraph F5e3 of this chapter.

---

[1]Will be the same only in case the state of the VP changed from "running" to "ready" and if this is the highest priority ready VP.

Finally, the ITC$ADVANCE calls VPSCHEDULER to schedule the next VP. Upon return from VPSCHEDULER, it will unlock the VPM.

J. KERNEL PROCESSES

The kernel processes make up the non-distributed kernel. Non-distributed here has the meaning that these processes are not distributed as part of each process's address space. Instead they represent system services and are used in the management of physical resources and execute asychronously with respect to user processes.

In this implementation all system processes are permanently bound to dedicated virtual processors, because it is very expensive to use a dedicated real processor.

Currently, two kernel processes are used, the Memory Management Process and the Idle Process (MMGT and IDLE Process respectively). The MMGT process controls both primary and secondary memory and the IDLE process defines the "no work" state of the system.

The currently implemented MMGT and IDLE processes do not have their final form. Instead they are "stubs" for these processes. The current implementation does provide the interface of these processes with the operating system and the inter-virtual processor synchronization mechanism, which is the most difficult task when implementing such processes. (This inter-virtual processor synchronization mechnism will

also be used in the future when Input/Output management will be added to the system.)

1. The Memory Management Process (MMGT Process)

The currently implemented MMGT process is permanently bound to the VP$START (see Figure 39) and the IDLE process is permanently bound to the VP$END. In this way these two virtual processors are in contention for physical processors but not for application (user) process scheduling.

Anderson [19] in his thesis describes the system-wide initialization. Below is described what is going on in each physical processor.

Each physical processor starts executing in the code of the ITC$INIT module (see paragraph I2 of this chapter). This module ends with a call to VPSCHEDULER. The VPSCHEDULER schedules the highest priority (i.e., VP$START) virtual processor to run on each physical processor. In this way each physical processor executes the MMGT process as its first process.

The MMGT process calls the loader module which repeatedly calls the CREATE$PROCESS module. When the loader is finished, the number of APT entries (processes) is equal to the number of application processes to be loaded. The module CREATE$PROCESS (see paragraph K10 in this chapter) initializes the address space (stacks) for each process and finally calls the module AWAITFORSTART (see paragraph K8). The result is that each newly created process becomes blocked

waiting for the special eventcount "START", with initial value zero, to reach the value 1.

When no other process remains to be loaded the MMGT process invokes ADVANCEFORSTART. The result is that the value of this special eventcount START reaches the value 1 and all the created processes on its blocked list (see Figure 44) are now awakened.

Then the MMGT process calls the module ITC$AWAIT (see paragraph I19 of this chapter). The result is that the VP$START enters the waiting state and finally the VPSCHEDULER is invoked. The VPSCHEDULER will schedule the VP which is loaded with the highest priority application process (one of the two central VP of Figure 39) since the VP$END is bound to the IDLE process. If no application processes are loaded on the specific physical processor, the VPSCHEDULER will schedule the lowest priority VP$END to run the IDLE process, since the highest priority MMGT process is currently blocked.

2. The Idle Process (IDLE Process)

The IDLE process defines the "no work" state of the system. The operating system attempts to schedule useful work on system processors whenever feasible. If there is no work then the IDLE process assures that the physical processor always has some valid process address space to execute in. The idle virtual processors act as "default" processors that will only be run when no other eligible VP is found.

Currently the IDLE process constitutes just an "idle loop". When the IDLE process is running, the this loop is first entered the current value of the PRDS software COUNTER (see paragraph H8 of this chapter) is obtained. Afterwards each time this idle loop is executed this COUNTER is updated (see also paragraphs I4 and I5).

By being able to read the value of these COUNTERS (one per physical processor) the performance of the operating system, the hardware communication links between different "clusters" and finally the effectiveness of the application processes "partitioning" can be actually tested.

The reason is, this COUNTER value records how much time each real processor executed in the IDLE process. These values can be interpreted and used as relative time or as actual time by multiplying the COUNTER's value by the time needed this idle loop to be executed once.

When in the future the preventive fault diagnosis and recovery routines are developed, part of these routines will be incorporated into the IDLE process, so that when a physical processor has no work it will execute this preventive fault diagnosis routine instead of idling.

K. THE TRAFFIC CONTROLLER

The Traffic Controller resides at level 2, multiplexes the user processes among virtual processor and manages the execution of these processes (process management) by invoking

the extended instructions of the virtual processors in level 1 (ITC-level). In addition to implementing the level 2 scheduling algorithm, the Traffic Controller creates the extended instruction set: TC$AWAIT and TC$ADVANCE.

TC$AWAIT and TC$ADVANCE (Traffic Controller AWAIT and ADVANCE) are used to implement an inter-process communication and synchronization mechanism invoked by the Supervisor, by using the eventcounts and sequencers.

The Traffic Controller's principal global data base (APT) has already been discussed in paragraph H2 of this chapter. Each entry of the APT corresponds to an application process and contains sufficient information to enable a virtual processor to be bound to and execute it.

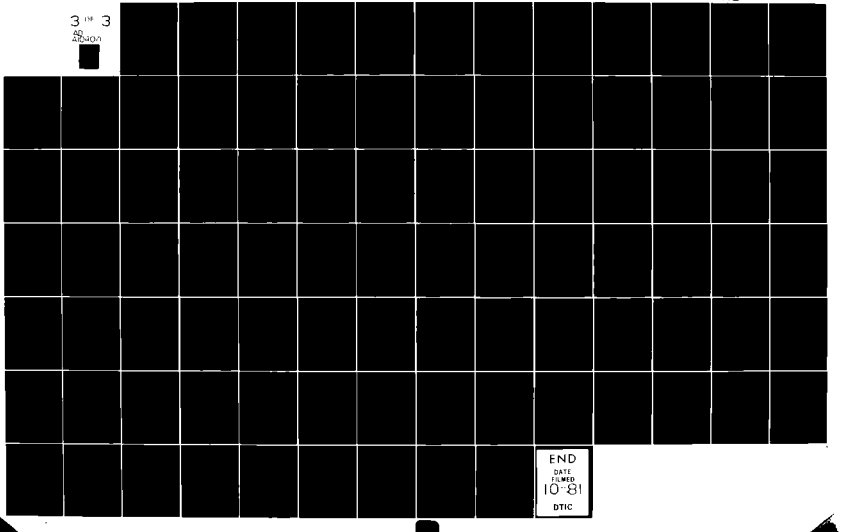
1. Process Scheduler (TC$SCHEDULER)

The TC$SCHEDULER works in essentially the same way that the Inner Traffic Controller's Scheduler (VPSCHEDULER) does. However, the TC$SCHEDULER schedules processes, while the VPSCHEDULER schedules virtual processors. The TC$SCHEDULER can be called by the TC$AWAIT, TC$ADVANCE, and TCPEHANDLER (Traffic Controller Preemption Handler).

It selects the highest priority ready process from the specific microcomputer's Loaded List (see Figure 44) to be bound to an available virtual processor. The TC$SCHEDULER works only with the processes which are runnable on its own physical processor using the fixed set of the four virtual processors assigned to this physical processor.

AD-A104 071 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
DETAILED DESIGN AND IMPLEMENTATION OF THE KERNEL OF A REAL-TIME--ETC(U)
MAR 81 D K RAMANTZIKOS

UNCLASSIFIED NL



When the TC$SCHEDULER finds a runnable process, the Inner Traffic Controller module ITC$LOAD$VP is called to bind the selected process to the running virtual processor. Alternatively, if there is no runnable process, the virtual processor will be idled (bound to the Idle Process and placed in the idle state) by a call to the Inner Traffic Controller module IDLE$VP.

2. Traffic Controller Locate Eventcount (TC$LOCATE$EVC)

This is a "utility" function called only by the Traffic Controller modules TC$AWAIT and TC$ADVANCE. Together with the following module TC$LOCATE$SEQ it is used to simplify the handling of eventcounts and sequencers respectively.

Its input argument is a pointer to the name of the eventcount. When invoked, TC$LOCATE$EVC makes a linear search in the Eventcount table (EVC$TABLE) to locate the desired eventcount by matching the names. If a match is found it returns the index of the specific eventcount in the EVC$TABLE in the Accumulator (AX) register, otherwise (if not found), it returns an error code.

3. Traffic Controller Locate Sequencer (TC$LOCATE$SEQ)

This is the second "utility" function used in the handling of sequencers and is called only by the TC$TICKET (Traffic Controller TICKET) module.

TC$LOCATE$SEQ works in exactly the same way as the LOCATE$EVC does except that it searches for sequencers in the Sequencer Table (SEQ$TABLE) instead of eventcounts in the EVC$TABLE.

4. Traffic Controller AWAIT (TC$AWAIT)

The TC$AWAIT is an inter-process synchronization primitive visible to the user, via the "GATE. It allows a process to suspend its own execution pending the occurrence of a specified event. TC$AWAIT is called with two input arguments, (a pointer to) the name of the eventcount and the value (of the event) to be awaited.

Upon invokation, Await locks the Active Process Table and then calls the Inner Traffic Controller utility function ITCRETVPTC to obtain the identity of the running virtual processor. This is used in a search of the Active Process Table to identify the process which invoked the TC$AWAIT.

Once the calling process has been identified, an implicit call is made to the TC$LOCATE$EVC to locate the index in the EVC$TABLE of the input argument (eventcount name). Then the current value of the eventcount kept in the EVC$TABLE is compared to the awaited value specified in the call. If the event has not yet occurred (viz., the current value in the EVC$TABLE is less than the awaited input argument value), then the process will enter the blocked state. The Value of Eventcount Awaited field in the Active Process Table is updated with the awaited argument value and the process is placed on the eventcount's Blocked List (see Figure 44). Otherwise, if the event has already occurred (viz., the current value is greater than or equal to the awaited input argument value), then the process is not blocked but is made ready.

Finally, in both cases, TC$AWAIT calls the TC$SCHEDULER to schedule the highest priority ready process. Upon the return from TC$SCHEDULER it unlocks the Active Process Table.

5. Traffic Controller ADVANCE (TC$ADVANCE)

The TC$ADVANCE is an inter-process synchronization primitive visible to the user, via the "GATE". It allows a process to signal the occurrence of an event. It updates the eventcount and signals those processes which had blocked themselves for this event. Thus TC$ADVANCE is also responsible for invoking the preemption mechanism.

TC$ADVANCE is called with one input argument, (a pointer to) the name of the eventcount being advanced.

It first locks the Active Process Table, then makes an implicit call to the TC$LOCATE$EVC to locate the index in the EVC$TABLE of the input argument (eventcount name). Then the current value of the eventcount in the EVC$TABLE is incremented by one. The eventcount's Blocked List (see Figure 44) is searched for processes which had previously blocked themselves waiting for the same eventcount to reach this value. As processes are found that should be awakened, viz., if the current value of the eventcount in the EVC$TABLE is greater or equal to the EVC$VALUE$AW (awaited eventcount value) field of the APT corresponding to the specific process, then these processes are made ready.

An entry in a temporary array of physical processors is now made to record the physical processor in whose local memory the newly awakened process is loaded for preemption. The awakened process is then removed from the eventcount's Blocked List.

Once all of the processes to be awakened have been found, TC$ADVANCE determines which virtual processors must be preempted. This is done for each of the previously flagged physical processors by first assuming that all of the physical processor's TC-visible virtual processors (two in this implementation) should be preempted. Then the decision is made as to which ones will not be preempted. This method greatly simplifies the algorithm. First a temporary list (array) of virtual processors is initialized to indicate a virtual preempt for each of the virtual processors. The Loaded List is then searched to find those processes which should be running. The processes which should be running are those with the highest priorities that are either in the "ready" or the "running" states. Assuming that there are 2 virtual processors per physical processor used for multiplexing, then the 2 highest priority "ready" or "running" processes in the Loaded List should be running. Any lower priority processes that actually are running should be preempted. TC$ADVANCE determines which of the processes that should be running already are running and deletes their virtual processors from the preemption list

(resets the preempt flag in this array). What will remain at the end are those virtual processors that are to be preempted.

The next step is to actually issue the preempt interrupts. The temporary preempt list is checked and if a preempt is indicated for a virtual processor, the Inner Traffic Controller module ITC$SEND$PREEMPT is called to actually issue the preempt.

TC$ADVANCE next readies the process which invoked it and calls the TC$SCHEDULER. Upon the return from the TC$-SCHEDULER the Active Process Table is unlocked.

6. Traffic Controller Ticket (TC$TICKET)

The routine TC$TICKET is also used in the inter-process synchronization and communication mechanism. It is the only operation performed on sequencers. It expects one input argument, (a pointer to) the sequencer name and it is visible to the user via the "GATE".

When invoked, TC$TICKET locks the Active Process Table, and calls implicitly the TC$LOCATE$SEQ to find the index in the global sequencer table (SEQ$TABLE) of the sequencer name given to it as the input argument. It then obtains from the SEQ$TABLE the current sequencer value (SEQ$VALUE) corresponding to the specific index and returns this sequencer's value to the process which called the TC$TICKER. The value according to the PL/M 86 language conventions is returned to the accumulator (AX) register.

Before returning, TC$TICKET increments by one the value of the sequencer and finally unlocks the Active Process Table.

In this way, TC$TICKET returns an unique sequencer value with every invokation, which will always be one more than the last value returned in the same way that TC$ADVANCE increments the eventcount value (EVC$VALUE). This is the reason why eventcounts and sequencers were defined as "positive non-decreasing integers".

7. Traffic Controller Preemption Handler (TCPEHANDLER)

The TCPEHANDLER is not a separate procedure but is just a label in the main program of the TC.

It serves as the virtual preempt interrupt entry point into TC$SCHEDULER and is invoked only by the Inner Traffic Controller Scheduler (VPSCHEDULER) in the course of virtualizing preempt interrupts. Actually the VPSCHEDULER transfers the program control, via the virtual interrupt vector, to the global label TCPEHANDLER. Recall that the virtual interrupt vector residing in the PRDS (VIRTINT-VECTOR) is initialized to point to the TCPEHANDLER label.

The TCPEHANDLER first locks the Active Process Table, then calls the TC$SCHEDULER which will find the highest priority ready process and bind it to the preempted virtual processor. Upon return from the TC$SCHEDULER the program control is transferred back to the VRSCHEDULER, effecting a "virtual interrupt return".

8. Await For Start (AWAITFORSTART)

This module is a part of TC$AWAIT and is called only once, during system initialization, by the MMGT process. It is invisible to the user.

It accepts three input arguments, the index of the process in the Active Process Table assigned by the CREATE$-PROCESS (Create Process) module discussed in paragraph 9 below, the eventcount name and the eventcount value to be awaited. There is in the system a special eventcount named "START" with initial value zero. The second input argument is the name of this special eventcount and the third, the awaited value, which is always one.

Each time the CREATE$PROCESS is called to create a process, the last statement is a call to AWAITFORSTART (Process, Start, 1). In this way each newly created process after creation is set to the blocked state awaiting for the special event START to reach the value 1. Each new process is added to the blocked list (see Figure 44) for the event-count START.

9. Advance For Start (ADVANCEFORSTAR$)

This module is a part of TC$ADVANCE and is also called only once during the system initialization by the MMGT process. It is invisible to the user.

It accepts one input argument (a pointer to) the name of the eventcount START. Where invoked it advances (incre-ments by one) the value of the special event START. The

result is that the value of START, initially zero, reaches for every new process the awaited value of 1. Then using the existing signalling mechanism, (the same as in the TC$ADVANCE module), ADVANCE$FOR$START awakes each process on the START eventcount's blocked list and sets its state to ready.

The created processes are now in contention for processor resources. The same sequence of actions will be followed as in the case of TC$ADVANCE except that ADVANCE$-FOR$START doesn't ready the calling process (which is the MMGT process) and also doesn't call the TC$SCHEDULER but merely returns program control to the caller, the MMGT process.

10. Create Process (CREATE$PROCESS)

The CREATE$PROCESS module provides the capability to dynamically create processes. It is called with one input argument, a pointer to a process parameter block (PPB) structure containing all the information necessary to initilize the process's stacks and enter the newly created process into the Active Process Table. All of the process' segments had previously been loaded into memory by the system loader, as described by Anderson [19].

CREATE$PROCESS first locks the Active Process Table. The next step is to enter the process in the Active Process Table. To create this entry the traffic controller uses the parameters passed by the PPB structure (see MMGT Process in

previous paragraph J1 of this chapter). The process is also inserted into the Load list based on its priority, viz., CREATE$PROCESS searches down the LOAD$LIST corresponding to the physical processor on which this process is loaded and sets the LOAD$THREAD field (see Figure 44) in such a way that the currently created process is entered immediately ahead of the first process found to have lower or equal priority.

Then CREATE$PROCESS initializes two stack frames for this process: the KERNEL$STACK and USER$STACK corresponding to the kernel and user domain respectively. In this way the process' address space is divided into these two separate domains of execution. The kernel stack has already been discussed in Paragraph E of this chapter (see also Figure 26). The user stack is shown in Figure 48 and the relation between these two stacks in Figure 49. Since in the PL/M-86 language the stack grows downwards (see Figure 22) by keeping the kernel stack above the user stack the KERNEL$STACK is protected from accidental user tampering (viz., overwriting KERNEL$-STACK is avoided).

The location of these stacks and the initial register values (viz., initial values for all of the 8086's registers) for the specific process are passed by the PPB structure and used in the initialization of the stack frames.

Finally, CREATE$PROCESS unlocks the Active Process Table and calls AWAITFORSTART (Await for Start) to block

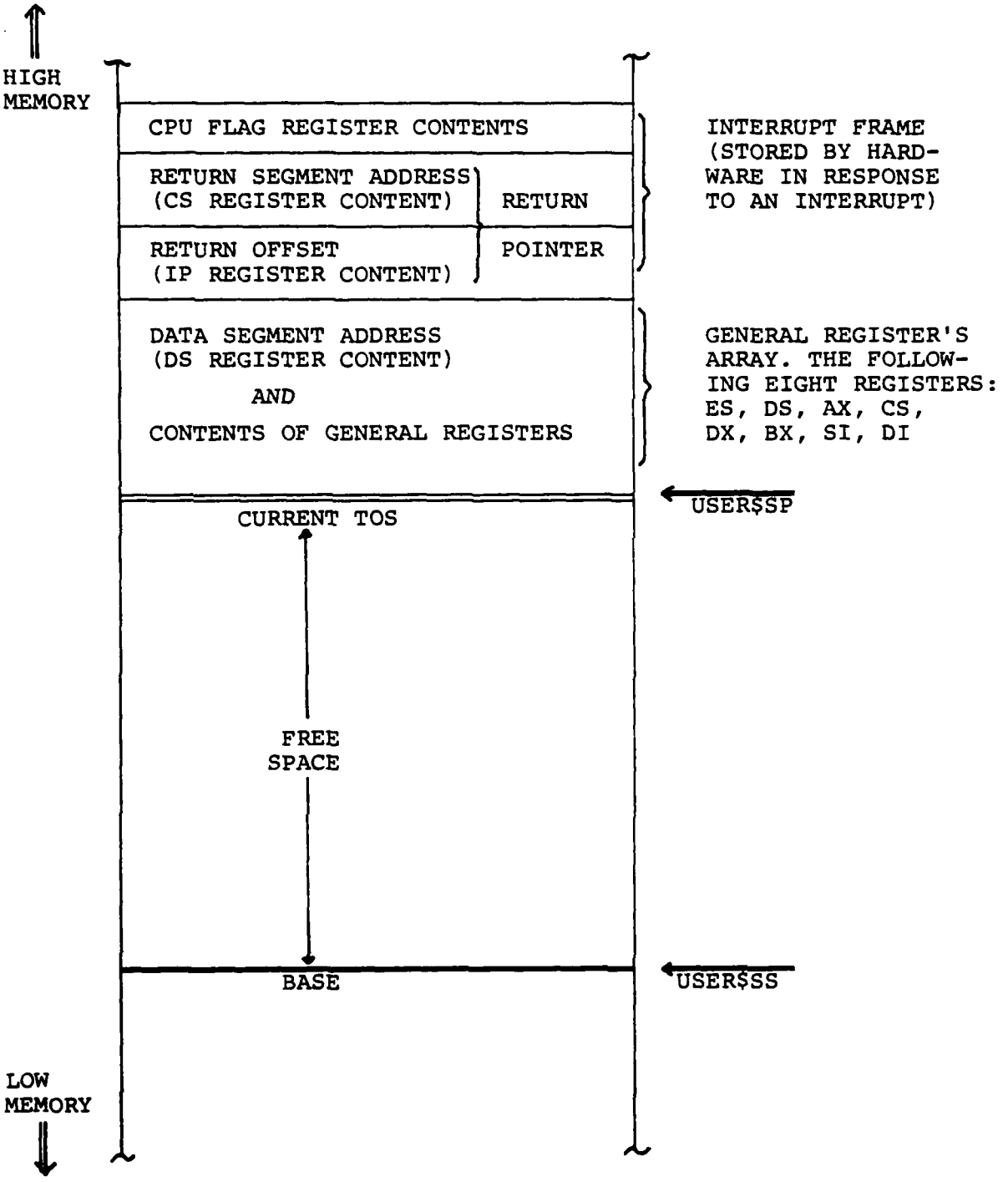


FIGURE 48. USER STACK AFTER RESPONDING TO AN INTERRUPT

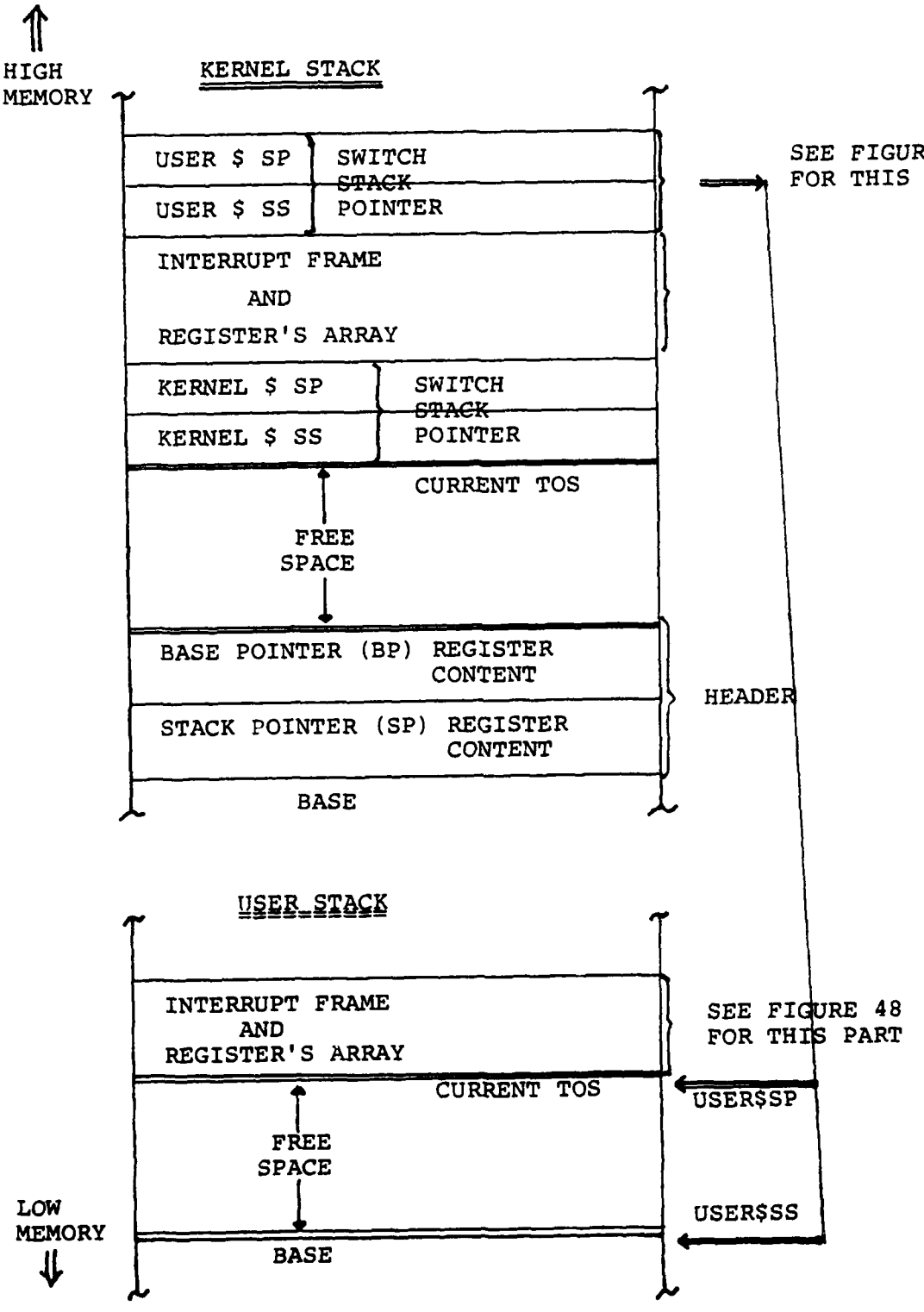


FIGURE 49. RELATION BETWEEN USER AND KERNEL STACKS

the newly created process and sets it in the blocked list of the special eventcount START.

11. Traffic Controller Create Eventcount (TC$CREATE$EVC)

This module is visible to the user via the "GATE". When invoked by an application process it creates the eventcount specified by this process. TC$CREATE$EVC is called with two input arguments, (a pointer to) the name of the eventcount to be created and the desired initial value, by the definition of eventcount [10] this value should always be zero.

Upon invokation, TC$CREATE$EVC locks the APT. It then calls TC$LOCATE$EVC to determine whether or not the eventcount had already been created. This is to avoid making duplicate entries (since each process which will use the eventcount must declare at least the name). If the eventcount had not previously been created (viz., no entry is found in the Eventcount Table with the same name as given in the input argument) then an entry is made in the Eventcount Table. The name is copied into the Eventcount Table EVC$NAME field and the eventcount's current value (EVC$VALUE field) is initialized to the second input argument. Otherwise no entry is made. When the entry is made in the Eventcount Table the APT$PTR field is initialized to FFH (the nil pointer), meaning that there is no process in the blocked list corresponding to this eventcount (empty blocked list).

The value of the variable EVENTS (see paragraph H3 of this chapter) is incremented by one each time an eventcount

is created. In this way the operating system keeps track of how many eventcounts are currently used.

Finally, TC$CREATE$EVC unlocks the APT and returns the program control to the calling procedure.

12. Traffic Controller Create Sequencer (TC$CREATE$SEQ)

This module is also visible to the user, via the "GATE". When invoked by an application process it creates a sequencer in exactly the same way that TC$CREATE$EVC creates an eventcount. The only difference is that it accepts one input argument, (a pointer to) the name of the sequencer as defined by the user. The initial sequencer value is always zero.

The operating system keeps track how many sequencers are currently used in the system by using the variable SEQUENCERS (see paragraph H9 of this chapter).

13. Traffic Controller Read (TC$READ)

The TC$READ module is also visible to the user, via the "GATE". It returns the current value of an eventcount to the calling process. It is called by one input argument, (a pointer to) the name of the specific eventcount.

When invoked, TC$READ locks the APT and then calls the TC$LOCATE$EVC to obtain the index of this eventcount in the eventcount table (EVC$TABLE). Using this index, TC$READ obtins the current value of the eventcount from the EVC$VALUE field of the EVC$TABLE and returns this value in the accumulator (AX) register.

Prior to returning to the calling procedure it unlocks the APT.

14. An Overall View Figure

After finishing the detailed description of the ITC, system processes and TC, an overall view of the two-level scheduling and multiplexing technique is illustrated in Figure 50. This view is similar for each of the physical processors in the system.

At the ITC level (LEVEL 1) the left most and right most virtual processor, e.g., VP$START and VP$END are permanently bound to the MMGT and IDLE process respectively. They are in contention for physical resources (in the figure for the physical processors), but they are not in contention for user process scheduling. The remaining two central VP's are temporarily bound to supervisor processes (user or application processes) as determined each time by the TC$SCHEDULER. The criteron is that the highest priority process will be scheduled first. In the case when no supervisor process is ready, the TC invokes ITC IDLE$VP (see paragraph I10) which loads an idle process on the VP. The idle process will actually run only when the VP to which it is permanently bound (VP$END) is scheduled. This will happen only when all other VP's are waiting the occurrence of events or temporarily bound to idle processes (i.e., when there is "no work" for the specific physical processor).

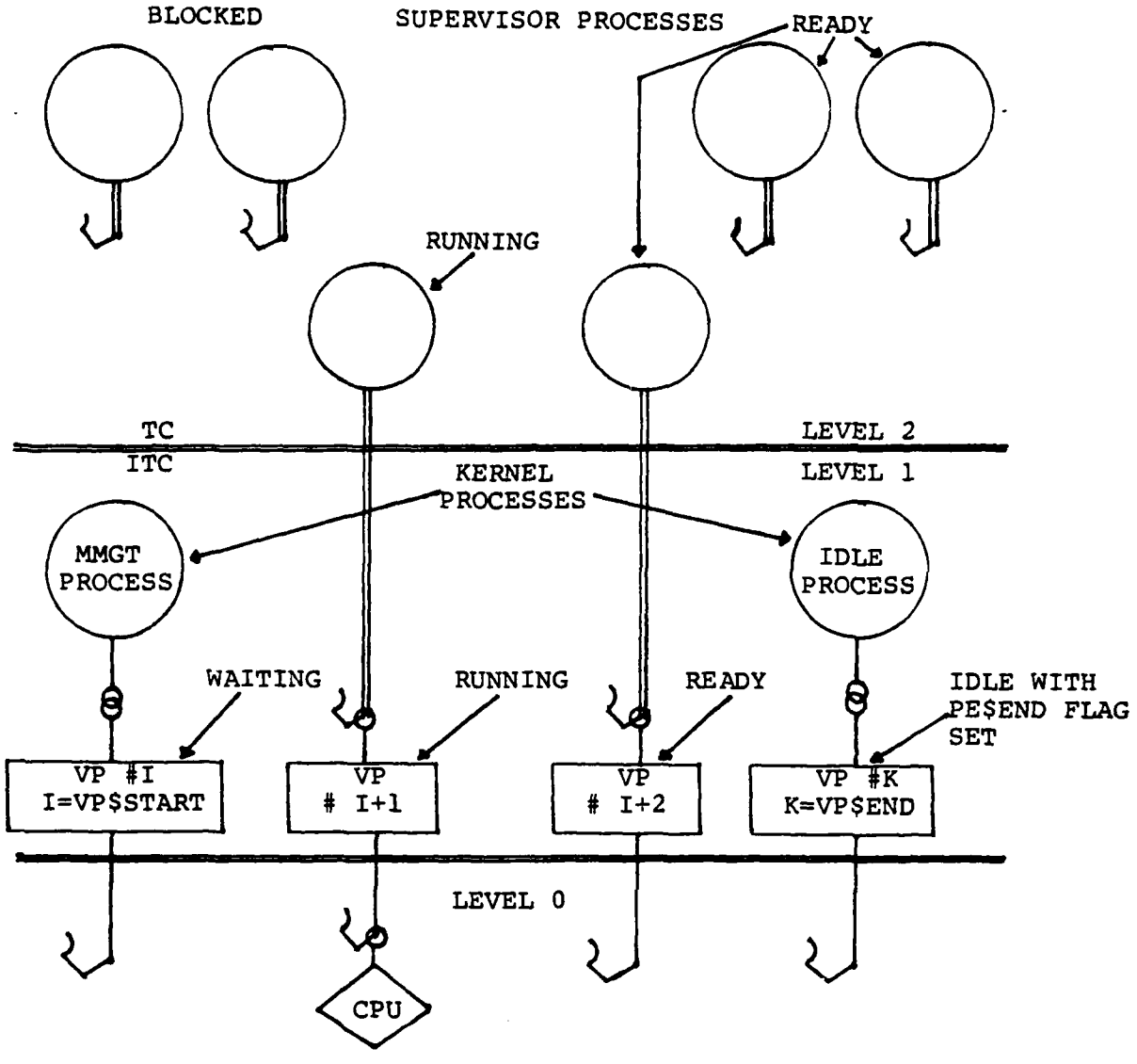


FIGURE 50. AN OVERALL VIEW FOR EACH PHYSICAL PROCESSOR

The ITC VPSCHEDULER schedules VP's on the physical processors. The criteron is that each time it schedules on the physical processor the highest priority "eligible" VP. Eligible in this design means ready or in the idle state, but with the preempt pending flag set.

In this way the operating system supports multiprogramming on each physical processor and also multiprocessing (concurrent processing) since there are several processors.

The transitions of the processer among the "ready-blocked-run" states is controlled by the inter-process communication and synchronization mechanism and also the TC$SCHEDULER.

The transitions of the VP's among the "idle-waiting-ready and run" states are controlled by the inter-virtual processor communication and synchronization mechanism and the VPSCHEDULER.

Finally, the hardware interrupt structure is used for preemptive scheduling to support real-time processing.

L. THE SUPERVISOR

1. General Description

In a general-purpose computer utility the "supervisor" provides the interface between application programs and the kernel of the operating system by supporting common services such as development tools (e.g., editors, compilers, assemblers, linkers, locaters, loaders), library functions, file system etc.

In the current implementation only one module is needed at the supervisor level, since all the above development tools are supported by the INTEL's MDS system (Microcomputer Development System). This module is written in assembly language and is called "Gate" or "Gatekeeper".

There must exist a way to link each user (application) program with the operating system in order to have as a result the user (application) process shown in Figure 25. The Gate is this "actual linkage" and is constructed such that it is the only operating system module that the user has to link to his program in order to access kernel functions visible to him.

2. The Gate or Gatekeeper

The Gate exists on the boundary between the kernel and supervisor levels of abstraction (see Figures 4, 5 and 9) and therefore is called a "software ring crossing mechanism". It is utilized to ensure that the kernel is "isolated" and "tamperproof". This module will be also important in the future if the system's internal security is considered. This structure is specifically designed to be compatible with the future version of the 8086 processor.

The system services visible to the user are: TC$AWAIT, TC$ADVANCE, TC$TICKET, TC$CREATEEVC, TCCREATE$SEQ and TC$READ. All these modules are related to the synchronization and communication mechanism. It is noted that the operating system never calls (execute the code of) these procedures. They are

called only by the user when the application programs need synchronization support (viz., when an application program is partitioned into asynchronous interactive parts).

The corresponding names for these procedures in the GATE are AWAIT, ADVANCE, TICKET, CREAT$EVC, CREATE$SEQ, and READ respectively.

The GATE contains the "public" declarations for these procedures and in this way allows the user to call these operating system procedures in exactly the same way that any other "external" procedure would be called.

The advantage is that only the GATE (a very small module) is required to be linked and loaded with each user process and not the entire operating system. Furthermore, during system generation [19], the GATE can be located in exactly the same absolute address in memory for all of the processes loaded on a single microcomputer. The result is that the GATE segment loaded in with each process will be overlayed and the same copy will be shared. This minimizes the amount of physical memory used by the GATE.

The GATE is a set of global procedures which the user programs can call directly. Each of the user accessible (visible) kernel functions is represented by one of these procedures. Actually they only set up the required parameters and use a "trap" feature (INT instruction) to effect the call to the real procedure of the kernel. For example, when a user program calls AWAIT then the GATE using the same parameters calls TC$AWAIT, and son on.

The GATE is written in assembly langauge because of the stack manipulation that must be done for parameters passing between PL/M 86 and ASM 86 (PL/M high level language and assembly language) and to invoke the "trap handler" in such a way to: 1) determine the correct kernel entry point (the proper procedure) to call, and 2) properly pass parameters to the kernel procedures.

The GATE consists of three small modules called GATE, trap handler and trap processes. When a GATE procedure is called by a user program the parameters are moved on the stack and the GATE reaches the trap handler by an interrupt (e.g., an internal interrupt, or trap) using the INT instruction. The trap handler transfer program control to the corresponding trap process which in turn invokes the real kernel procedure with the same parameters passed on the stack by the user program.

This has the effect of de-coupling the user from all the operating system modules below the Supervisor level.

The software provided by the Gatekeeper has to perform additional functions upon the kernel entry and kernel exit, as shown in Figure 51.

Figure 52 tabulates the required format for all of the external procedure declarations that must be included in the user programs when invoking kernel functions. Of course, only the kernel functions actually invoked need to be externally declared by the user program.

Kernel Entry

1. Mask hardware preempt interrupts in the kernel.
2. Save user domain registers in the user stack (user domain).
3. Switch from user to kernel domain (stack).
4. Save user domain stack segment (SS) register and user stack pointer (SP) register in the kernel stack.
5. Check arguments and invoke appropriate kernel entry point.

Kernel Exit

1. Check for virtual preempt interrupts (call CHECK$PREEMPT) when leaving the kernel (unmask virtual interrupt).
2. Save kernel domain SS and SP registers in the kernel stack.
3. Restore user domain SS and SP registers.
4. Restore user domain registers.
5. Unmask hardware interrupts.
6. Return to the user process, execution point in the user domain.

FIGURE 51. KERNEL ENTRY - KERNEL EXIT

Creating an Eventcount:

```
CREATE$EVC:  PROCEDURE(EVENTCOUNT, VALUE) EXTERNAL;
             DELARE EVENTCOUNT POINTER, VALUE WORD;
END;
```

Creating a Sequencer:

```
CREATE$SEQ:  PROCEDURE(SEQUENCER) EXTERNAL;
             DECLARE SEQUENCER POINTER;
END;
```

The Advance Operation:

```
ADVANCE:  PROCEDURE(EVENTCOUNT) EXTERNAL;
          DECLARE EVENTCOUNT POINTER;
END;
```

The Await Operation:

```
AWAIT:  PROCEDURE(EVENTCOUNT,VALUE) EXTERNAL;
        DECLARE EVENTCOUNT POINTER,
                VALUE WORD;
        END;
```

The Ticket Operation:

```
    TICKET:  PROCEDURE(SEQUENCER) BYTE EXTERNAL;
             DECLARE SEQUENCER POINTER;
    END;
```

The Read Operation:

```
    READ:  PROCEDURE(EVENTCOUNT) BYTE EXTERNAL;
           DECLARE EVENTCOUNT POINTER;
    END;
```

FIGURE 52. KERNEL CALL EXTERNAL PROCEDURE DECLARATIONS

In J. Wasson thesis [8], there is a whole appendix (Appendix A) of 33 pages with programming instructions and examples how to use the synchronization mechanism and the operating system. It is considered redundant to repeat these instructions. Instead, in Appendix A of this thesis will be incorporated several actual operating system test programs and their output.

V. CONCLUSIONS

A. RESULTS

The principal goal of this thesis, the development of the kernel of a real-time, distributed operating system for a microcomputer based multiprocessor system was met.

This operating system is hierarchically structured, layered in three loop free levels of abstraction, viz., the Inner Traffic Controller, the Traffic Controller and the Supervisor, and fundamentally configuration independent.

This verifiable loop free structure was demonstrated with EXAMPLE #6 in the Appendix A.

Furthermore, at each level of this hierarchical structure the corresponding part of the operating system consists of a set of understandable modules whose interactions are clearly specified and strictly enforced.

The result is a relatively small and easy to analyze operating system and this also was a principal goal.

Since the kernel is small: (1) less memory is spent for its storage and (2) less processor time is spent in its execution. This advantage of less memory allows physical distribution of the kernel's code and data among the microcomputers and this distribution in turn helps to minimize system bus contention.

On the other hand the layered modular structure provides the advantage of making it easy to debug, test and analyze, ensuring correct operation and permitting an opportunity to increase performance by tuning.

B. FOLLOW ON RESEARCH

Although the kernel executes correctly, as shown in the examples of appendix A, before higher levels of abstraction are added to the system, a more formal test and evaluation plan should be developed. Once the kernel has been proven highly reliable then the follow on research is feasible for the reasons explained below.

The existing stub for memory management process solves the two hardest problems of the memory management functions: (1) the interface with the kernel and (2) the needed inter-virtual processor communication and synchronization. Both capabilities have been implemented and tested.

The hard problem for adding I/O management is also the inter-virtual synchronization mechanism which exists and works correctly. For I/O management one more VP will be added on each physical processor permanently bound to the I/O process, in the same way as is done for MMGT and IDLE process.

It is also possible to add file management (by dividing its functions among kernel and Supervisor). Finally, the process oriented structure of the operating system, the

separation of the address space of each process into user and kernel domain of execution and also the existence of the Gate lead automatically to the required structure for internal security. Additional segmentation hardware is needed to control the access (read, write) of the system's and user's subjects (viz., processes), to the system objects (viz., segments). The needed hardware will be available in the anticipated 8086 successor.

APPENDIX A

SYSTEM'S TESTING

This appendix incorporated six examples to demonstrate the use of the operating system and also to test the inter-process communication and synchronization mechanism, the inter-virtual processor communication and synchronization mechanism and the inter-real processor communication mechanism (used for preemptive scheduling and supported by the hardware interrupt structure).

For each example, the input source code and the actual output to the printer are incorporated.

Four of these test programs designed and implemented by the author and the remaining two by students working in the "Electro-Optics and Signal Processing Laboratory" of the Naval Postgraduate School.

EXAMPLE #1

In this example there are two interactive processes running on a uniprocessor system under the operating system. This example demonstrates the multiprogramming capability and also the use of the inter-process communication and synchronization mechanism.

In the input source code under the header EXAMPLE #1 INPUT, there are enough comments for easy understanding of this example. Figures 53 and 54 are provided to illustrate the interleaved execution of these two processes and how they interact using the synchronization mechanism. The output on the printer is also provided under the header EXAMPLE #1 OUTPUT.

The variables A, B and C have been incorporated and are changed before and after entering the operating system kernel (e.g., when the process calls ADVANCE or AWAIT) to demonstrate that these values are correctly saved and restored from the per process stack.

EXAMPLE #1 INPUT

```
/* FILE PR1.SRC    JANUARY 29  1981                                    */

PR1$MODULE:  DO;

DECLARE (A,B,C,I) WORD,
        DISPLAY BYTE;
DECLARE MSG1(*) BYTE INITIAL ('ENTERING PROC#1, IT HAS HIGHER PRIORITY ',
        MSG2(*) BYTE INITIAL ('PROC#1, ENTERING DELAY  '),
        MSG3(*) BYTE INITIAL ('EXECUTING IN PROC#1  '),
        MSG4(*) BYTE INITIAL ('END OF DEMONSTRATION  '),
        MSG5(*) BYTE INITIAL ('CURRENT VALUE OF  C =  '),
        CR  LITERALLY '0DH',
        LF  LITERALLY '0AH';
DECLARE DELTA(6) BYTE DATA('DELTA%'),
        OMEGA(6) BYTE DATA('@MEGA%');

AWAIT: PROCEDURE (EVC$ID$PARM,EVC$VAL$PARM) EXTERNAL;
       DECLARE EVC$ID$PARM POINTER,
               EVC$VAL$PARM WORD;
END;

ADVANCE: PROCEDURE (EVC$ID$PARM) EXTERNAL;
         DECLARE EVC$ID$PARM POINTER;
END;

OUT$CHAR: PROCEDURE(CHAR);
          DECLARE CHAR BYTE;
          DO WHILE (INPUT(0DAH) AND 01H) = 0;
          END;
          OUTPUT(0D8H) = CHAR;
END;

OUT$HEX: PROCEDURE(B);
         DECLARE B BYTE;
         DECLARE ASCII(*) BYTE DATA('0123456789ABCDEF');
         CALL OUT$CHAR(ASCII(SHR(B,4) AND 0FH));
         CALL OUT$CHAR(ASCII(B AND 0FH));
END;

/******** ======> M A I N  P R O G R A M  <====== ********/

/* PROC#1 IS THE HIGHER PRIORITY PROCESS */

CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);
CALL OUT$CHAR(LF);
DO I = 0 TO 39;
   CALL OUT$CHAR(MSG1(I));
END;
CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);

B = 5;
A = B*5+25;
C = A*10;

CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);
CALL OUT$CHAR(LF);
DO I = 0 TO 22;
   CALL OUT$CHAR(MSG2(I));
END;
CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);
```

```
DO I = 0 TO 200;
   CALL TIME(250);
END;

/* THE INITIAL VALUE OF EVENTS DELTA AND OMEGA ARE 0 */

CALL AWAIT(3DELTA,2);

/* BECAUSE THE VALUE IN THE CALL STATEMENT IS 2, GREATER THAN THE */
/* INITIAL VALUE = 0, AWAIT WILL SUSPEND THE EXECUTION OF THAT    */
/* (CALLING) PROCESS AND THE SCHEDULER WILL SCHEDULE THE HIGHER   */
/* PRIORITY PROC FROM THE REMAINING PROCESSES. PROC#1 WILL GO TO THE */
/* BLOCKED STATE */

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 0 TO 19;
   CALL OUTSCHAR(MSG3(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 0 TO 22;
   CALL OUTSCHAR(MSG2(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

DO I = 0 TO 200;
   CALL TIME(250);
END;

A = 0;
C = A*2;
A = C*5;

CALL AWAIT(3OMEGA,1);

/* SINCE 1 IS GREATER THAN PRESENT VALUE OF OMEGA = 0, PROC#1 */
/* WILL GO AGAIN TO THE BLOCKED STATE AND SCHEDULER WILL SCHE-*/
/* DULE PROC#2 */

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 0 TO 19;
   CALL OUTSCHAR(MSG3(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 0 TO 22;
   CALL OUTSCHAR(MSG2(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

DO I = 0 TO 200;
```

```
   CALL DIVF(352);
END;

I = A/B;

CALL OUTSCHAR(LF);
DO I = 0 TO 22;
   CALL OUTSCHAR(MSG5(I));
END;
DISPLAY = HIGH(C);
CALL OUTSHEX(DISPLAY);
DISPLAY = LOW(C);
CALL OUTSHEX(DISPLAY);
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);

CALL AWAIT(3DELTA,3);

/* ONCE MORE THIS PROCESS GOES TO THE BLOCKED STATE, SINCE THE PRESENT */
/* VALUE OF DELTA IS 2, ( 2 < 3 ). */

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 2 TO 19;
   CALL OUTSCHAR(MSG3(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 0 TO 22;
   CALL OUTSCHAR(MSG4(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

END;  /* OF PP1SMODULE */
```

```
/* FILE PR2.SRC    JANUARY 05  1981                                    */

PR2$MODULE:  DO;

DECLARE (A,B,C) INTEGER,
        I WORD;
DECLARE MSG1(*) BYTE INITIAL ('ENTERING PROC#2. IT HAS LOWER PRIORITY '),
        MSG2(*) BYTE INITIAL ('PROC#2. ENTERING DELAY '),
        MSG3(*) BYTE INITIAL ('EXECUTING IN PROC#2 '),
        CR  LITERALLY '0DH',
        LF  LITERALLY '0AH';
DECLARE DELTA(0) BYTE DATA('DELTA%'),
        OMEGA(0) BYTE DATA('OMEGA%');

$WAIT: PROCEDURE (EVC$ID$PARM,EVC$VAL$PARM) EXTERNAL;
       DECLARE EVC$ID$PARM POINTER,
               EVC$VAL$PARM WORD;
END;

ADVANCE: PROCEDURE (EVC$ID$PARM) EXTERNAL;
         DECLARE EVC$ID$PARM POINTER;
END;

OUT$CHAR: PROCEDURE(CHAR);
          DECLARE CHAR BYTE;
          DO WHILE (INPUT(0DAH) AND 01H) = 0;
          END;
          OUTPUT(0DEH) = CHAR;
END;


/******** ======>  M A I N   P R O G R A M  <======  ********/

/* PROC#2 IS THE LOWER PRIORITY PROCESS */

CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);
CALL OUT$CHAR(LF);
DO I = 0 TO 38;
   CALL OUT$CHAR(MSG1(I));
END;
CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);

A = 12;
B = A*12;
C = B+A;

CALL ADVANCE(@DELTA);

/* EVENT DELTA HAS NOW THE VALUE 1, BUT PROC#1 CONTINUES SLEEPING */
/* UNTIL THE EVENT DELTA WILL REACH THE VALUE 2 */

CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);
CALL OUT$CHAR(LF);
DO I = 0 TO 22;
   CALL OUT$CHAR(MSG2(I));
END;
CALL OUT$CHAR(CR);
CALL OUT$CHAR(LF);

DO I = 0 TO 200;
```

```
   CALL TIME(250);
END;

/* THE INITIAL VALUE OF EVENTS DELTA AND OMEGA ARE 2 */

C = C-10;

CALL ADVANCE(.DELTA);

/* EVENT DELTA REACHES VALUE 2, SO ADVANCE WILL AWAKE PROC#1, AND */
/* ALSO WILL SET PROC#2 FROM RUN TO THE READY STATE. THEN THE     */
/* SCHEDULER WILL SCHEDULE THE HIGHER PRIORITY REMAINING PROCESS  */
/* (IN OUR CASE PROC#1).                                          */

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 0 TO 19;
   CALL OUTSCHAR(MSG3(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 2 TO 22;
   CALL OUTSCHAR(MSG2(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

DO I = 0 TO 200;
   CALL TIME(250);
END;

C = C+50;
B = C*2;
A = B*5;

CALL ADVANCE(.OMEGA);

/* THE EVENT OMEGA REACHES NOW THE VALUE 1, SO ADVANCE WILL   */
/* AWAKE PROC#1 AND SET PROC#2 IN READY STATE. THEN THE  SCHE-*/
/* DULER WILL SCHEDULE PROC#1 (SINCE BOTH PROCESSES ARE NOW IN*/
/* THE READY STATE AND PROC#1 HAS HIGHER PRIORITY.            */

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 2 TO 19;
   CALL OUTSCHAR(MSG3(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);
CALL OUTSCHAR(LF);
DO I = 0 TO 22;
   CALL OUTSCHAR(MSG2(I));
END;
CALL OUTSCHAR(CR);
CALL OUTSCHAR(LF);

DO I = 0 TO 200;
```

```
   CALL TIME(C60);
END;

B = A**2;

CALL ADVANCE(DELTA);

/* EVENT DELTA REACHES NOW VALUE 3, SO PROC=1 AWAKES AND IS SCHEDULED  *
/* TO RUN. AND SO ON. */

END;  /* OF PRGSMODULE */
```

EXAMPLE #1 OUTPUT

```
ENTERING PROC#1. IT HAS HIGHER PRIORITY

PROC#1. ENTERING DELAY
ENTERING    A W A I T

ENTERING PROC#2. IT HAS LOWER PRIORITY
ENTERING A D V A N C E

PROC#2. ENTERING DELAY
ENTERING A D V A N C E

EXECUTING IN PROC#1

PROC#1. ENTERING DELAY
ENTERING    A W A I T

EXECUTING IN PROC#2

PROC#2. ENTERING DELAY
ENTERING A D V A N C E

EXECUTING IN PROC#1

PROC#1. ENTERING DELAY
CURRENT VALUE OF  C =  03E8

ENTERING    A W A I T

EXECUTING IN PROC#2

PROC#2. ENTERING DELAY
ENTERING A D V A N C E
```

EXECUTING IN PROC#1

END OF DEMONSTRATION

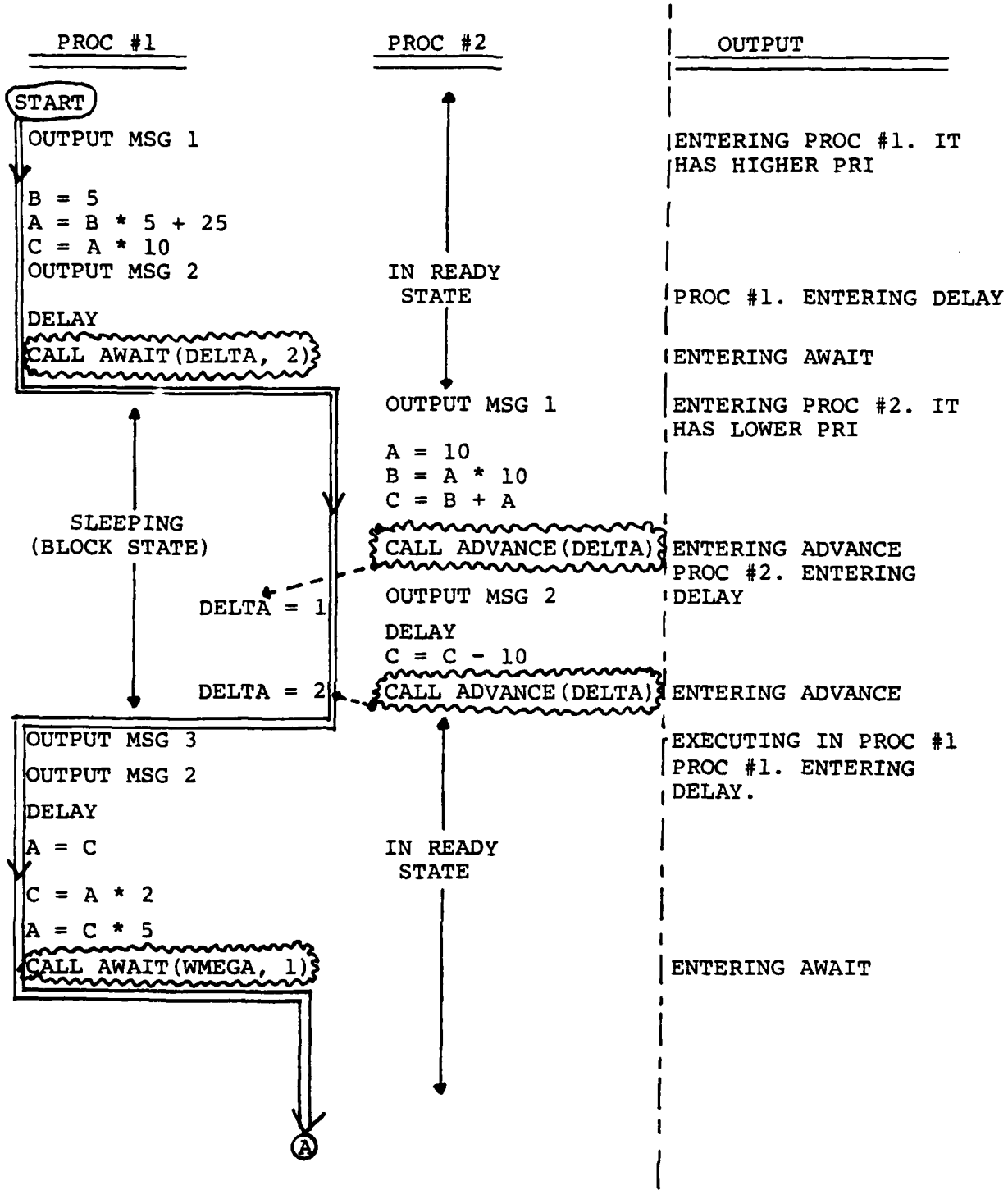


FIGURE 53. INTERLEAVED EXECUTION OF TWO PROCESSES

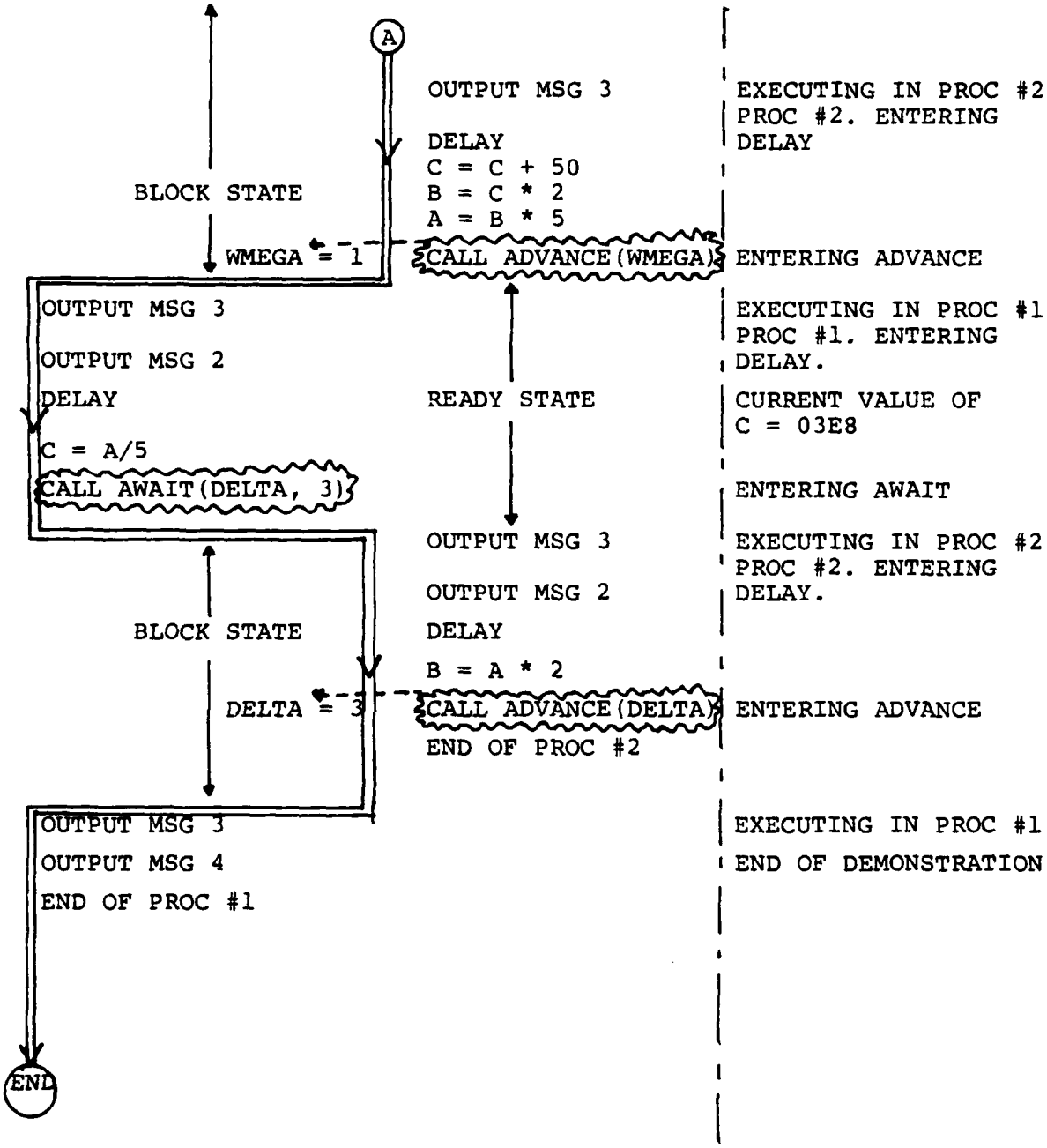


FIGURE 54. INTERLEAVED EXECUTION OF TWO PROCESSES

EXAMPLE #2

In this example there are two interactive processes running on a uniprocessor under the operating system. These two processes simulate the image processing processes CLUTTER SUPPRESION AND FILTER DESIGN. The data comes into the micro-computer as frames of images and an extensive use of the synchronization mechanism is required.

Following the comments in the input source code under the header EXAMPLE #2 INPUT and the output messages under the header EXAMPLE #2 OUTPUT, it is possible to follow the interleaving execution and interaction of these two processes.

EXAMPLE #2 INPUT

```
      /*       FILE    PROC01.SRC      16 MAY   */

CSUPPSMODULE: DO;

     DECLARE    I BYTE;
     DECLARE    CR  LITERALLY '0DH',
                LF  LITERALLY '0AH';

     DECLARE J BYTE;

     DECLARE CSUPP(6) BYTE DATA('CSUPP%');
     DECLARE FLDES(6) BYTE DATA('FLDES%');

DECLARE
 MSG1(*) BYTE INITIAL ('PROC#1. INITIAL ENTRY INTO CLUTTER SUPPRESSION '),
 MSG2(*) BYTE INITIAL ('PROC#1. WAIT FOR DATA READY                    '),
 MSG3(*) BYTE INITIAL ('PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: '),
 MSG4(*) BYTE INITIAL ('PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT       ');

     AWAIT: PROCEDURE(EVC$ID$PARM,EVC$VAL$PARM) EXTERNAL;
        DECLARE EVC$ID$PARM  POINTER,
                EVC$VAL$PARM WORD;
     END;

     ADVANCE: PROCEDURE(EVC$ID$PARM) EXTERNAL;
        DECLARE EVC$ID$PARM POINTER;
     END;

     OUT$CHAR: PROCEDURE(CHAR);
        DECLARE CHAR BYTE;
        DO WHILE (INPUT(0DAH) AND 01H) = 0; END;
        OUTPUT(0D8H) = CHAR;
     END;

     OUT$HEX: PROCEDURE(B);
        DECLARE B BYTE;
        DECLARE ASCII(*) BYTE DATA ('0123456789ABCDEF');
        CALL OUT$CHAR(ASCII(SHR(B,4) AND 0FH));
        CALL OUT$CHAR(ASCII(B AND 0FH));
     END;

     I = 0;

        CALL OUT$CHAR(LF);

        DO Z = 0 TO 46;
          CALL OUT$CHAR(MSG1(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);
        CALL OUT$CHAR(LF);

     DO WHILE (I <= 64);

        CALL OUT$CHAR(LF);
        DO Z = 0 TO 46;
          CALL OUT$CHAR(MSG2(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);
        CALL OUT$CHAR(LF);
```

```
            CALL AWAIT(@CSUPP,1);

            I = I + 1;

            CALL OUT$CHAR(LF);
            DO Z = 0 TO 48;
              CALL OUT$CHAR(MSG3(Z));
            END;
            CALL OUT$BEEP(1);
            CALL OUT$CHAR(CR);
            CALL OUT$CHAR(LF);
            CALL OUT$CHAR(LF);

            DO T = 0 TO 200;
              CALL TIME(250);
            END;

            CALL OUT$CHAR(LF);
            DO Z = 0 TO 46;
               CALL OUT$CHAR(MSG4(Z));
            END;
            CALL OUT$CHAR(CR);
            CALL OUT$CHAR(LF);
            CALL OUT$CHAR(LF);

            CALL ADVANCE(@FLDES);

         END; /* WHILE */

      END; /* MODULE */
```

```
/*      FILE    PROC02.SRC      18 MAY   */

FLDESSMODULE: DO;

     DECLARE    I BYTE;
     DECLARE    CR  LITERALLY '0DH',
                LF  LITERALLY '0AH';

     DECLARE Z BYTE;

     DECLARE CSUPP(6) BYTE DATA('CSUPP%');
     DECLARE FLDES(6) BYTE DATA('FLDES%');

DECLARE
 MSG1(*) BYTE INITIAL ('PROC#2. INITIAL ENTRY INTO FILTER DESIGN '),
 MSG2(*) BYTE INITIAL ('PROC#2. AWAIT FOR DATA READY '),
 MSG3(*) BYTE INITIAL ('PROC#2. PERFORMING FILTER DESIGN ON FRAME: '),
 MSG4(*) BYTE INITIAL ('PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT ');

     AWAIT: PROCEDURE(EVC$ID$PARM,EVC$VAL$PARM) EXTERNAL;
        DECLARE EVC$ID$PARM  POINTER,
                EVC$VAL$PARM WORD;
     END;

     ADVANCE: PROCEDURE(EVC$ID$PARM) EXTERNAL;
        DECLARE EVCSID$PARM POINTER;
     END;

     OUT$CHAR: PROCEDURE(CHAR);
        DECLARE CHAR BYTE;
        DO WHILE (INPUT(2DAH) AND 01H) = 0; END;
        OUTPUT(0D8H) = CHAR;
     END;

     OUT$HEX: PROCEDURE(B);
        DECLARE B BYTE,
                ASCII(*) BYTE DATA ('0123456789ABCDEF');
        CALL OUT$CHAR(ASCII(SHR(B,4) AND 0FH));
        CALL OUT$CHAR(ASCII(B AND 0FH));
     END;

     I = 0;

        DO Z = 0 TO 40;
          CALL OUT$CHAR(MSG1(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);
        CALL OUT$CHAR(LF);

        DO WHILE (I <= 64);

        CALL OUT$CHAR(LF);
        DO Z = 0 TO 29;
          CALL OUT$CHAR(MSG2(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);
        CALL OUT$CHAR(LF);

        CALL AWAIT(@FLDES,I);
```

```
            I = I + 1;

            DO Z = 0 TO 200;
               CALL TIME(250);
            END;

            CALL OUT$CHAR(LF);
            DO Z = 0 TO 42;
              CALL OUT$CHAR(MSG3(Z));
            END;
            CALL OUT$HEX(I);
            CALL OUT$CHAR(CR);
            CALL OUT$CHAR(LF);
            CALL OUT$CHAR(LF);

            CALL OUT$CHAR(LF);
            DO Z = 0 TO 46;
              CALL OUT$CHAR(MSG4(Z));
            END;
            CALL OUT$CHAR(CR);
            CALL OUT$CHAR(LF);
            CALL OUT$CHAR(LF);

            CALL ADVANCE(@CSJPP);

         END; /* WHILE */

      END; /*MODULE */
```

EXAMPLE #2 OUTPUT

```
PROC#1. INITIAL ENTRY INTO CLUTTER SUPPRESSION

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 01

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T
PROC#2. INITIAL ENTRY INTO FILTER DESIGN

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 01

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 02

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T
```

```
PROC#2. PERFORMING FILTER DESIGN ON FRAME: 02

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 03

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 03

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 04

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 04

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT
```

```
ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 05

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 05

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 06

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 06

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 07
```

```
PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 07

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 08

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E

PROC#1. WAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. AWAIT FOR DATA READY

ENTERING    A W A I T

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 08

PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING A D V A N C E

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 09

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING A D V A N C E
```

EXAMPLE #3

The input source code for this example is exactly the same as for the previous example. The difference is the output. The output under the header EXAMPLE #3 OUTPUT has more output messages. In fact in every module of the operating system has been incorporated at least one output message as shown in Figure 55.

In this way the debugging and checking becomes easier. Also it is possible to follow the flow of program control between several modules of the operating system.

The higher priority process is PROCØ1 (CLUTTER SUPPRESSION) with priority 40 and the lower priority is PROCØ2 (FILTER DESIGN) with priority 41.

The address space descriptor for the first process (the base of its "per process stack") is equal to 6000H(this appears as 600 because of INTEL's monitor convention), for the second it is 7000H and for the idle process it is 5000.

LEVEL	MODULE	OUTPUT MESSAGE
	CHECKVIRTINT	ENTERING CHECKVIRTINT
	ITCRETVP	ENTERING ITCRETINT RUNNINGVPID =
	ITC$LOAD$VP	ENTERING ITC$LOAD$VP LOADING VP NUMBER: PRIORITY FOR THIS VP IS: NEW DBR FOR THIS VP IS:
ITC	CHECK$PREEMPT	ENTERING CHECKPREEMPT
	GETWORK	ENTERING GETWORK SELECTED$DBR =
	RUN$THIS$VP	ENTERING RUNTHISVP SET VP TO RUNNING: VP =
	ITC$SEND$PREEMPT	ENTERING ITC$SEND$PREEMPT
	LOCKVPM	ENTERING LOCKVPM
	UNLOCKVPM	ENTERING UNLOCKVPM
	RDYTHISVP	ENTERING RDYTHISVP SET UP TO READY: VP =
	TC$SCHEDULER	ENTERING TC$SCHEDULER
	TC$LOCATE$EVC	ENTERING TC$LOCATE$EVC
TC	TC$AWAIT	ENTERING AWAIT
	TC$ADVANCE	ENTERING ADVANCE
	TCPEHANDLER	ENTERING TCPEHANDLER

FIGURE 55. OUTPUT MESSAGES OF THE OPERATING SYSTEM'S MODULES

EXAMPLE #3 OUTPUT

```
ENTERING CHECKVIRTINT
ENTERING LOCKVPM
ENTERING RDYTHISVP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  00
   SET VP TO READY:  VP =  00
ENTERING GETWORK
            SELECTED$DBR =  0500
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING TC$PE$HANDLER
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
      LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:  40
  NEW DBR FOR THIS VP IS:  0600
ENTERING GETWORK
            SELECTED$DBR =  0600
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01

PROC#1. INITIAL ENTRY INTO CLUTTER SUPPRESSION


PROC#1. WAIT FOR DATA READY


ENTERING    A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING TC$LOCATE$EVC
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
      LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:  40
  NEW DBR FOR THIS VP IS:  0600
ENTERING GETWORK
            SELECTED$DBR =  0600
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 01
```

PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

```
ENTERING A D V A N C E

ENTERING TC$LOCATE$EVC
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
       LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:   40
  NEW DBR FOR THIS VP IS:   0600
ENTERING GETWORK
            SELECTED$DBR =  0600
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#1. WAIT FOR DATA READY


ENTERING    A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING TC$LOCATE$EVC
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
       LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:   41
  NEW DBR FOR THIS VP IS:   0700
ENTERING GETWORK
            SELECTED$DBR =  0700
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
PROC#2. INITIAL ENTRY INTO FILTER DESIGN


PROC#2. AWAIT FOR DATA READY


ENTERING    A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING TC$LOCATE$EVC
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
       LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:   41
```

```
 NEW DBR FOR THIS VP IS: 0700

ENTERING GETWORK
           SELECTED$DBR =  0700
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 01


PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT


ENTERING A D V A N C E

ENTERING TC$LOCATE$EVC
ENTERING ITC$SEND$PREEMPT
ENTERING ITC$SEND$PREEMPT
ENTERING ITC$RET$VP
           RUNNING$VP$ID =  01
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
           RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
           RUNNING$VP$ID =  01
      LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:  40
  NEW DBR FOR THIS VP IS:  0600
ENTERING GETWORK
           SELECTED$DBR =  0600
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 02


PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT


ENTERING A D V A N C E

ENTERING TC$LOCATE$EVC
ENTERING ITC$RET$VP
           RUNNING$VP$ID =  01
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
           RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
           RUNNING$VP$ID =  01
      LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:  40
  NEW DBR FOR THIS VP IS:  0600
ENTERING GETWORK
           SELECTED$DBR =  0600
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#1. WAIT FOR DATA READY


ENTERING    A W A I T

ENTERING ITC$RET$VP
           RUNNING$VP$ID =  01
ENTERING TC$LOCATE$EVC
```

```
ENTERING ITC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
       LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:   41
  NEW DBR FOR THIS VP IS:   0700
ENTERING GETWORK
            SELECTED$DBR =  0700
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#2. AWAIT FOR DATA READY


ENTERING  . A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING TC$LOCATE$EVC
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
       LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:   41
  NEW DBR FOR THIS VP IS:   0700
ENTERING GETWORK
            SELECTED$DBR =  0700
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#2. PERFORMING FILTER DESIGN ON FRAME: 02


PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT


ENTERING A D V A N C E

ENTERING TC$LOCATE$EVC
ENTERING ITC$SEND$PREEMPT
ENTERING ITC$SEND$PREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING TC$SCHEDULER
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOAD$VP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
       LOADING VP NUMBER:   01
 PRIORITY FOR THIS VP IS:   40
  NEW DBR FOR THIS VP IS:   0600
ENTERING GETWORK
            SELECTED$DBR =  0600
ENTERING RUNTHISVP
 SET VP TO RUNNING:  VP =  01

PROC#1. PERFORMING CLUTTER SUPPRESSION ON FRAME: 03
```

EXAMPLE #4

This example was designed and implemented by Kurt Holmquist for testing purposes of the operating system, and consist of five processes running on a uniprocessor system under the operating system.

In the following pages are included the input source code under the header EXAMPLE #4 INPUT and the output of the microcomputer directly to the printer under the header EXAMPLE #4 OUTPUT.

EXAMPLE #4 INPUT

```
/*  This module contains a program from which
    multiple processes can be generated by
    the D/S synchronization and scheduling
    functions.   The number of processes
    to be created is the value "num_proc".
    The processes should all be in the "ready"
    state initially.                              */

MULTI_PROC:  DO;

/******  External procedure declarations  ******/

Error_key:  PROCEDURE (value) EXTERNAL;
    DECLARE  value  WORD;
END;

Read_char:  PROCEDURE  BYTE  EXTERNAL;
END;

Message:  PROCEDURE  (mesg_ptr)  EXTERNAL;
    DECLARE  mesg_ptr  POINTER;
END;

Await:  PROCEDURE  (event_name,count)  EXTERNAL;
    DECLARE  event_name  POINTER,
             count       WORD;
END;

Advance:  PROCEDURE  (event_name)  EXTERNAL;
    DECLARE  event_name  POINTER;
END;

/******  Event count declaration  ******/

DECLARE  advance_next  (5) BYTE DATA ('ADNXT%');

/******  Message and variable declarations  ******/

DECLARE

m_prun (*)  BYTE INITIAL (2DH,0AH,0AH,'RUNNING PROCESS #'),
pnum   (*)  BYTE INITIAL ('0   DBR = ',0),
m_adva (*)  BYTE INITIAL (0DH,0AH,'
                          HIT A KEY TO CALL "ADVANCE".',2),
m_awai (*)  BYTE INITIAL (0DH,0AH,'
                          HIT A KEY TO CALL "AWAIT".',0),

(awaited_count, ss_reg, process) WORD,
1 WORD INITIAL (0), char  BYTE;

DECLARE  num_proc  LITERALLY  '5';

DECLARE  ENTRY  LABEL  PUBLIC;

/*  Program for any number of processes to execute   */

ENTRY:  DO WHILE 1;

/*  Each process identifies itself by reading the
    value of the stack segment register (DBR).       */

    ss_reg = STACKBASE;
    process = SHR(ss_reg, 4) AND 0FH;
    pnum (0) = LOW (process) + 030H;
```

```
        i = i + 1;

/*  Print out the number of the process currently
    running and the TOD.                                    */

        CALL  Message (@m_prun);
        CALL  Output_hex (ss_tod);

        CALL  Message (@m_adva);

        char = Read_char;           /* Use keyboard input to step
                                       through the program.     */

/*  Because the last process on the load list has lower
    priority than all the others, it has a slightly different
    program sequence.  For the last process only, the process
    switch will take place when the call to Advance is made.  */

        IF process = num_proc THEN DO;

            CALL  Advance (@advance_next);

            END;

/*  All processes except the last one on the load list
    execute the following.                                  */

        ELSE DO;

/*  Advance the event count  */

            CALL  Advance (@advance_next);

            CALL  Message (@m_awai);

            char = Read_char;

            awaited_count = ((i-1)/5 + 1)*5;

            CALL  Await (@advance_next,awaited_count);

/*  The currently running process will become blocked
    at this point and another proces will begin running.
    When a process which has previously blocked itself
    begins running again, the entry point will be here
    and the following call will determine the process
    switch time.                                            */

            END;

        END;

END;
```

EXAMPLE #4 OUTPUT

```
RUNNING PROCESS #1    DBR = 0710
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #2    DBR = 0720
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #3    DBR = 0730
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #4    DBR = 0740
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #5    DBR = 0750
                              HIT A KEY TO CALL "ADVANCE".

RUNNING PROCESS #1    DBR = 0710
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #2    DBR = 0720
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #3    DBR = 0730
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #4    DBR = 0740
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #5    DBR = 0750
                              HIT A KEY TO CALL "ADVANCE".

RUNNING PROCESS #1    DBR = 0710
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #2    DBR = 0720
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #3    DBR = 0730
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #4    DBR = 0740
                              HIT A KEY TO CALL "ADVANCE".
                              HIT A KEY TO CALL "AWAIT".

RUNNING PROCESS #5    DBR = 0750
                              HIT A KEY TO CALL "ADVANCE".
```

EXAMPLE #5

This example was designed and implemented by the following students of Naval Postgraduate School:

LT Kenneth Webb
LCDR Leo Schnieder
LT Antony Christian

in partial fullfillment (as a project) of the requirements of the course CS 3550. It can be used to test the synchronization and communication mechanisms of the operating system.

In Figure 56 is shown the interactions of five processes: I/O CONTROLLER, ID-POSIT, CORRELATION, TRACK and DISPLAY. Also shown are five shared buffers: SENSBF, SENSDR, ACTIVE-BUFFER, OUTPUT TABLE and TRACK TABLE residing in the system's global memory while the processes code and data are located into the microcomputer's local (on board) memory. For example, the shared buffer SENSBF is used by the I/O CONTROLLER and ID-POSIT processes and so on.

Flow of information into and out of the various processes and buffers is indicated in Figure 56 by the direction of the arrows. Eventcounts are shown between the processes.

The details about this project will not be incorporated here since these can be extracted from the following input source code, under the header EXAMPLE #5 INPUT. The output to the printer follows the input source code, under the header EXAMPLE #5 OUTPUT.

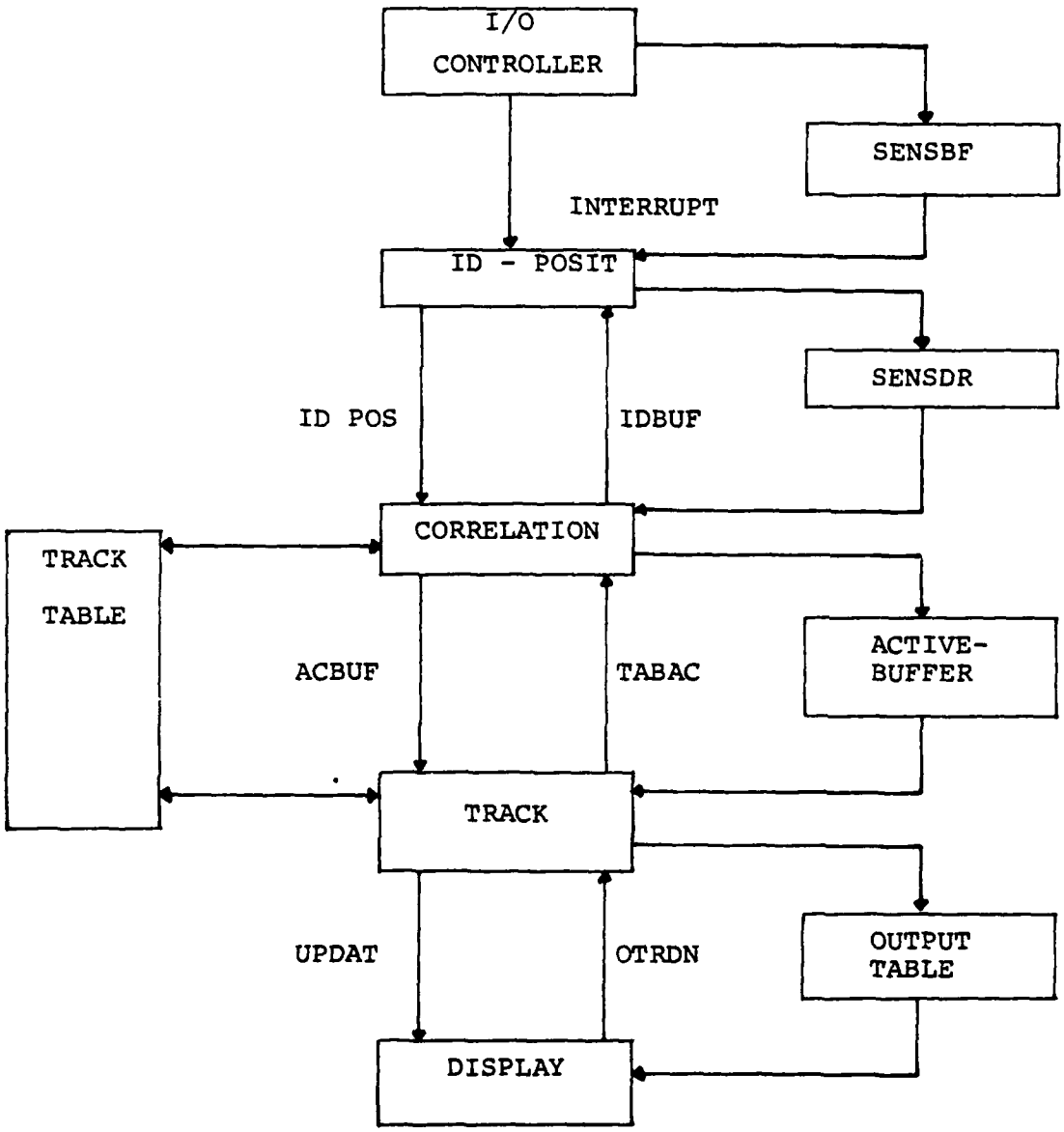


FIGURE 56. AN EXAMPLE OF FIVE INTERACTIVE PROCESSES

EXAMPLE #5 INPUT

```
IDPOSIT:DO; -

  /**MODULE BEGINNING*/


/* DECLARATIONS*/

DECLARE # WORD INITIAL(0);
DECLARE IDBUF (6) BYTE DATA ('IDBUF%');
DECLARE IDPOS (6) BYTE DATA ('IDPOS%');
DECLARE IDPRE (6) BYTE DATA ('IDPRE%');

DECLARE TGTSDATA(9) BYTE DATA('TGTSDATA%');
DECLARE ID$POSIT$START(15) BYTE DATA('ID$POSIT$START%');
DECLARE I INTEGER INITIAL (0);
DECLARE SENSBUF (20) STRUCTURE (INFO(15) BYTE, TYME WORD,
        FLAG$7 BYTE) EXTERNAL;
DECLARE RESET LITERALLY '00H';
DECLARE ZZ WORD initial (1);
DECLARE (J,L,K,BEARING) INTEGER;
DECLARE (X$SENS, Y$SENS, RANGE) INTEGER;
DECLARE BUFFER(15) INTEGER;
DECLARE (RNG$MULT$PTR, POSIT$PTR) POINTER;
DECLARE SENS$NUM1 INTEGER EXTERNAL;
DECLARE (X$TGT, Y$TGT) INTEGER EXTERNAL;
DECLARE TYME1 WORD EXTERNAL;
DECLARE (RNG$MULT BASED RNG$MULT$PTR)(2) INTEGER;
DECLARE (POSIT BASED POSIT$PTR)(2) INTEGER;


A#AIT: PROCEDURE (EVC$ID$PARM, EVC$VAL$PARM) EXTERNAL;
     DECLARE EVC$ID$PARM POINTER;
     DECLARE EVC$VAL$PARM WORD;
END A#AIT;


ADVANCE: PROCEDURE (EVC$ID$PARM) EXTERNAL;
       DECLARE EVC$ID$PARM POINTER;
END ADVANCE;

BNGANALYZER: PROCEDURE (BEARING) POINTER EXTERNAL;
             DECLARE BEARING INTEGER;
END BNGANALYZER;

TGT$POSIT: PROCEDURE (X$BASE, Y$BASE, X$COMP, Y$COMP, RNG) POINTER
           EXTERNAL;
        DECLARE (X$BASE,Y$BASE, X$COMP, Y$COMP, RNG) INTEGER;
END TGT$POSIT;


write: procedure (ptr) external;
declare ptr pointer;
end;


INITV2: PROCEDURE EXTERNAL;
END;


/*END OF THE DECLARATIONS*/


CALL INITV2;
```

```
   DO while 01;
      CALL AWAIT (SIIPRE,ZZ);
      ZZ = ZZ + 1;

      IF SENSBUF(I).FLAG$7 = 01H THEN

/*INCREMENT THE BUFFER COUNT AND OBTAIN THE DATA. CONVERT
  FROM ASCII TO NUMERICAL REPRESENTATION.*/

         IF I < 20 THEN
         DO;
            DO J = 0 TO 14;
               BUFFER(J) = INT(SENSBUF(I).INFO(J) - 30H);
            END;
         END;
         ELSE DO;
            I = 0;
            DO J = 0 TO 14;
               BUFFER(J) = INT( SENSBUF(I).INFO(J) - 30H);
            END;
         END;
         DO;
         SENSBUF(I).FLAG$7 = RESET;
         I = I + 1;
         END;


/* THE NEXT STEP IS TO CONSOLIDATE THE INPUT INFORMATION
   INTO THE APPROPRIATE VARIABLES*/

         DO;
            XSSENS =    ((1000 * BUFFER(1)) +
                         (100 * BUFFER(2)) +
                         (10 * BUFFER(3)) +
                         (BUFFER(4)));
            YSSENS =    ((1000 * BUFFER(5)) +
                         (100 * BUFFER(6)) +
                         (10 * BUFFER(7)) +
                         (BUFFER(8)));
            BEARING =    ((100 * BUFFER(9)) +
                          (10 * BUFFER(10)) +
                          (BUFFER(11)));
            RANGE =    ((100 * BUFFER(12)) +
                        (10 * BUFFER(13)) +
                        (BUFFER(14)));
         END;


/* CALL THE SUBROUTINES WHICH ANALYZE THE DATA TO PRODUCE
   THE TARGET'S POSITION ON AN X-Y GRID*/

         DO;
call write (3('Calling bearing_analizer.%'));
call write (3('Calling target_posit.%'));
         END;


/* LOAD THE MEMORY SHARED WITH THE CORRELATION PROCESS*/

         CALL AWAIT(SIDBUF,W);
 W = W + 1;
         DO;
    SENSSNUM1 = BUFFER(0);
```

```
            TIME1 = SENSEUP(I-1).TIME;
            XSTGT = POSIT(0);
            YSTGT = POSIT(1);
         END;


/* SET THE ADVANCE TO SIGNAL THE END OF THIS PROCESSOR RUN
   TO THE SCHEDULER*/



      CALL ADVANCE(@IDPOS);
END;
END IDsPOSIT;
```

```
CORRELATION:  DO;  /* BEGINNING OF MODULE */

/*  THIS MODULE DETERMINES WHETHER ANY INCOMING TARGETS CAN BE
    CORRELATED WITH ANY TRACKS ALREADY BEING TRACKED IN THE
    TRACKSTABLE.  IF THERE IS A CORRELATION,THEN THE MODULE
    UPDATES THE TARGET'S X POSITION, Y POSITION, AND TIME.
    IF THERE IS NO CORRELATION,THEN THE MODULE ASSIGNS THE
    NEW TARGET A NEW TRACK NUMBER AND ENTERS IT'S TRACK
    NUMBER, X POSITION, Y POSITION, AND TIME IN THE TRACKS
    TABLE.  */



            /* EXTERNAL DECLARATIONS */

    DECLARE TRACK$TABLE(25)STRUCTURE(TRACK$NR #ORD,X$POSIT INTEGER,
            Y$POSIT INTEGER,TYME #ORD,CSE INTEGER,SPD INTEGER,
            X$FUPOS INTEGER,Y$FUPOS INTEGER)EXTERNAL;

    DECLARE ACTIVE$BUFF STRUCTURE(TRACK$NR #ORD,X$POSIT INTEGER,
            Y$POSIT INTEGER,TYME #ORD)EXTERNAL;

    DECLARE SENS$NUM1 INTEGER EXTERNAL;
    DECLARE (FSTGT,FSTGT)INTEGER EXTERNAL;
    DECLARE TYME1 #ORD EXTERNAL;

             /* INTERNAL DECLARATIONS */

    DECLARE (N,J,FFOUND) #ORD;
    DECLARE Z #ORD;
    DECLARE W #ORD INITIAL(0);
    DECLARE M #ORD INITIAL(0);
    DECLARE (TRUE,FALSE)#ORD INITIAL(0FFH,00H);
    DECLARE IDPOS(6)BYTE DATA('IDPOS%');
    DECLARE ACBUF(6)BYTE DATA('ACBUF%');
    DECLARE IDBUF(6)BYTE DATA('IDBUF%');
    DECLARE TABAC(6)BYTE DATA('TABAC%');
    DECLARE (SENSOR,XTAR,YTAR)INTEGER;
    DECLARE TME #ORD;
    DECLARE MSG1(*)BYTE INITIAL('ENTERING CORRELATION%');
    DECLARE MSG2(*)BYTE INITIAL('LEAVING CORRELATION%');

            /* EXTERNAL PROCEDURES */

    /*  THE ADVANCE MODULE ADVANCES THE VALUE OF EVC$ID$PARM. */

    ADVANCE: PROCEDURE(EVC$ID$PARM) EXTERNAL;
             DECLARE EVC$ID$PARM POINTER;
             END ADVANCE;

/*  THE AWAIT MODULE WILL BLOCK THE CALLING PROGRAM FROM
    EXECUTION UNTIL EVC$ID$PARM=EVC$VAL$PARM.  */

    AWAIT: PROCEDURE(EVC$ID$PARM,EVC$VAL$PARM)EXTERNAL;
           DECLARE EVC$ID$PARM POINTER;
           DECLARE EVC$VAL$PARM #ORD;
           END AWAIT;

/*  THE XMATCH MODULE DETERMINES WHETHER THE INCOMING TARGET'S
    X POSITION CORRELATES WITH ANY FUTURE X POSITIONS OF TRACKS
    THAT ARE ALREADY IN THE TRACKSTABLE.                    */

    XMATCH: PROCEDURE(TABLESPTR,XTAR,N)#ORD EXTERNAL;
```

```
            DECLARE TABLE$PTR POINTER;
            DECLARE (XTAR INTEGER;
            DECLARE N WORD;
            END XMATCH;

/*  THE YMATCH MODULE COMPARES THE Y FUTURE POSITION OF THE TRACK
    FOUND IN XMATCH MODULE TO THE INCOMING TARGET'S Y POSITION
    AND DETERMINES IF THERE IS A CORRELATION.                    */

    YMATCH: PROCEDURE(FUTUREY,YTAR)WORD EXTERNAL;
            DECLARE FUTUREY INTEGER;
            DECLARE YTAR INTEGER;
            END YMATCH;

/*  THE TABLE MODULE FILLS THE TRACK$TABLE WITH TRACK NUMBER,
    X POSITION, Y POSITION, AND TIME OF ALL NEW TRACKS.      */

    TABLE: PROCEDURE(N,XTAR,YTAR,T)EXTERNAL;
           DECLARE N WORD;
           DECLARE (XTAR,YTAR)INTEGER;
           DECLARE T WORD;
           END TABLE;

/*  THE BUFF MODULE UPDATES THE ACTIVE$BUFF WITH TRACK NUMBER,
    X POSITION, Y POSITION, AND TIME OF ALL OLD TRACKS.      */

    BUFF: PROCEDURE(TN,XTAR,YTAR,T)EXTERNAL;
          DECLARE (TN,T)WORD;
          DECLARE (XTAR,YTAR)INTEGER;
          END BUFF;

/*  THE WRITE MODULE IS USED TO PRINT OUT MESSAGES. */

    WRITE: PROCEDURE(PTR)EXTERNAL;
           DECLARE PTR POINTER;
           END WRITE;


               /*  MAIN PROGRAM */


    DO Z=1 TO 10000;

          CALL WRITE(@MSG1);
          CALL AWAIT(@IDPOS,Z);

          SENSOR=SENS$NUM1;
          XTAR=X$TGT;
          YTAR=Y$TGT;
          TME=TYME1;

          CALL ADVANCE(@IDBUF);

          CALL AWAIT(@TABAC,W);

          YFOUND=FALSE;
          J=2;
          N=3;

          DO WHILE J<25 AND YFOUND=FALSE;


             J=XMATCH(@TRACK$TABLE,XTAR,N);
```

```
            IF J<25 THEN
            DO;
              FOUND=XMATCH(TRACK$TABLE(J).XSPOS,XTAR);
              IF XFOUND=TRUE THEN
              DO;
                 CALL BUFF(TRACK$TABLE(J),TRACK$NR,XTAR,YTAR,TME);
              END;

              ELSE
                 N=J+1;
            END;

         END;

         IF J=25 THEN
         DO;

            CALL TABLE(M,XTAR,YTAR,TME);
            M=M+1;

         END;

         CALL ADVANCE(SACBUF);
         N=N+1;
         CALL WRITE(SMSG2);


END;

                  END CORRELATION;  /* END OF MODULE */
```

```
TEST: DO:

DECLARE SET LITERALLY '01H',
OUTSTABLESPTR POINTER,
DELIMITER LITERALLY '25H';
declareUPDAT (6) BYTE DATA ('UPDAT%'),
OTRDY (6) BYTE DATA ('OTRDY%'),
acbuf (6) byte data ('acbuf%'),
tabac (6) byte data ('tabac%'),
# word initial (0),
1 word initial (0),
entry_a label public,
J word initial (1);


DECLARE BUFF (4) STRUCTURE (TRACK$NR WORD, X$POSIT INTEGER,
Y$POSIT INTEGER, TIME INTEGER, CSE INTEGER,
SPD INTEGER, FLAG BYTE);

DECLARE OUTPUT$TABLE STRUCTURE (TRACK$NR WORD, X$POSIT
INTEGER, Y$POSIT INTEGER, TIME INTEGER,
CSE INTEGER, SPD INTEGER, FLAG BYTE)
EXTERNAL;


WRITE: PROCEDURE (PTR) EXTERNAL;
DECLARE PTR POINTER;
END;

ADVANCE: PROCEDURE (EVC$ID$PARM) EXTERNAL;
DECLARE EVC$ID$PARM POINTER;
END ADVANCE;

AWAIT: PROCEDURE (EVC$ID$PARM, EVC$VAL$PARM) EXTERNAL;
DECLARE EVC$ID$PARM POINTER,
EVC$VAL$PARM WORD;
END;


entry_a:
CALL WRITE (@('BEGIN TEST OF DISPLAY.% '));


/* INITIALIZE  */

BUFF(1).TRACK$NR = 1;
BUFF(1).X$POSIT = 531;
BUFF(1).Y$POSIT = 1054;
BUFF(1).TIME = 134;
BUFF(1).CSE = 240;
BUFF(1).SPD = 345;
BUFF(1).FLAG = SET;

BUFF(3).TRACK$NR = 13;
BUFF(3).X$POSIT = 210;
BUFF(3).Y$POSIT = 2413;
BUFF(3).TIME = 248;
BUFF(3).CSE = 198;
BUFF(3).SPD = 1000;
BUFF(3).FLAG = SET;
```

```
TO while 01;
call await (@acbuf, w);
w = w + 1;
call write (@('Extracting data from a buffer shared with CORRELATE.%'));call
 advance (@tapac);


CALL AWAIT (@DTRDY, I);

OUTPUT$TABLE.TRACK$NR = BUFF(J).TRACK$NR;
OUTPUT$TABLE.X$POSIT = BUFF(J).X$POSIT;
OUTPUT$TABLE.Y$POSIT = BUFF(J).Y$POSIT;
OUTPUT$TABLE.TIME = BUFF(J).TIME;
OUTPUT$TABLE.CSE = BUFF(J).CSE;
OUTPUT$TABLE.SPD = BUFF(J).SPD;
OUTPUT$TABLE.FLAG = BUFF(J).FLAG;
I = I + 1;
CALL WRITE(@('BUFFER FILLED. ADVANCING DISPLAY.%'));

CALL ADVANCE (@UPDAT);
END;


END TEST;
```

```
DISPLAY: DO;



ADVANCE: PROCEDURE (EVCSIDSPARM) EXTERNAL;
DECLARE EVCSIDSPARM POINTER;
END;

AWAIT: PROCEDURE (EVCSIDSPARM, EVCSVALSPARM) EXTERNAL;
DECLARE EVCSIDSPARM POINTER,
EVCSVALSPARM WORD;
END;

WRITE: PROCEDURE (PTR) EXTERNAL;
DECLARE PTR POINTER;
END;


CONVERT$TABLE: PROCEDURE (TABLESPTR) POINTER EXTERNAL;
DECLARE TABLESPTR POINTER;
END;



DECLARE I WORD INITIAL (1),
PTR POINTER,
OUT$TABLES$PTR POINTER,
SET LITERALLY '01H',
RESET LITERALLY '00H',
DELIMITER LITERALLY '25H';
declareUPDAT (6) BYTE DATA ('UPDAT%'),
OUPDY (6' BYTE DATA ('OUPDY%'),
(J, K) BYTE;

DECLARE OUTPUTSTABLE STRUCTURE (TRACK$NR WORD, X$POSIT INTEGER,
Y$POSIT INTEGER, TIME INTEGER, CSE INTEGER,
SPD INTEGER, FLAG BYTE) EXTERNAL;



DECLARE LOC$BUFF STRUCTURE (TRACK$NR WORD, X$POSIT INTEGER,
 Y$POSIT INTEGER, TIME INTEGER, CSE INTEGER,
 SPD INTEGER, FLAG BYTE);

DECLARE (OUT$TABLE BASED OUT$TABLE$PTR) (25) STRUCTURE (TRK$NR
(10) BYTE, X$P (10) BYTE,
Y$P (10) BYTE, T (10) BYTE, CSE (10) BYTE, SPD
(10) BYTE, DELIM BYTE);


CALL WRITE (.('ENTERING DISPLAY.%'));


DO while 01;
CALL AWAIT (.UPDAT, I);
CALL WRITE(.('ENTERING DISPLAY LOOP.%'));
I = I + 1;
```

```
LOC$BUFF.TRACK$NR = OUTPUT$TABLE.TRACK$NR;
LOC$BUFF.X$POSIT = OUTPUT$TABLE.X$POSIT;
LOC$BUFF.Y$POSIT = OUTPUT$TABLE.Y$POSIT;
LOC$BUFF.TIME = OUTPUT$TABLE.TIME;
LOC$BUFF.CSE = OUTPUT$TABLE.CSE;
LOC$BUFF.SPD = OUTPUT$TABLE.SPD;
LOC$BUFF.FLAG = OUTPUT$TABLE.FLAG;


/* CONVERT DATA TO ASCII FOR OUTPUT.  */

out$table$ptr = convert$table (@loc$buff);


/* WRITE DATA ON CONSOLE.  */

CALL WRITE (@('TRACK_NR  X_POSIT   Y_POSIT   TIME      COURSE    SPEED%'));
do j = 0 to 25;
if out$table(J).ielln = ielimiter
then call write (@out$table(j));
end;



CALL ADVANCE (@OTRDY);


END;
END DISPLAY;
```

EXAMPLE #5 OUTPUT

```
Entering IDPOSIT LOOP.

ENTERING    A W A I T

ENTERING CORRELATION

ENTERING    A W A I T

BEGIN TEST OF DISPLAY.

ENTERING    A W A I T

ENTERING DISPLAY.

ENTERING    A W A I T

Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
CALL ADVANCE(ID$POSIT).

ENTERING A D V A N C E

Calling bearing_analizer.
Calling target_posit.

ENTERING    A W A I T


ENTERING A D V A N C E

Entering IDPOSIT LOOP.

ENTERING    A W A I T


ENTERING A D V A N C E


ENTERING    A W A I T


ENTERING A D V A N C E

LEAVING CORRELATION
ENTERING CORRELATION

ENTERING    A W A I T

Extracting data from a buffer shared with CORRELA   TE.

ENTERING A D V A N C E


ENTERING    A W A I T

BUFFER FILLED. ADVANCING DISPLAY.

ENTERING A D V A N C E
```

```
ENTERING    A W A I T

ENTERING DISPLAY LOOP.
TRACK_NR  X_POSIT   Y_POSIT   TIME      COURSE    SPEED
00        231.8     678.9     0067      240       0345
01        443.1     444.4     0175      240       0345

ENTERING A D V A N C E


ENTERING    A W A I T

Entering IDLE PROCESS.
Entering IDLE PROCESS.
CALL ADVANCE(ID$POSIT).

ENTERING A D V A N C E

Calling bearing_analizer.
Calling target_posit.

ENTERING    A W A I T


ENTERING A D V A N C E

Entering IDPOSIT LOOP.

ENTERING    A W A I T


ENTERING A D V A N C E


ENTERING    A W A I T


ENTERING A D V A N C E

LEAVING CORRELATION
ENTERING CORRELATION

ENTERING    A W A I T

Extracting data from a buffer shared with CORRELA   TE.

ENTERING A D V A N C E


ENTERING    A W A I T

BUFFER FILLED. ADVANCING DISPLAY.

ENTERING A D V A N C E


ENTERING    A W A I T

ENTERING DISPLAY LOOP.
TRACK_NR  X_POSIT   Y_POSIT   TIME      COURSE    SPEED
00        231.8     678.9     0067      240       0345
01        443.1     444.4     0107      240       0345

ENTERING A D V A N C E
```

```
ENTERING    A W A I T

Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
EnterinCALL ADVANCE(ID$POSIT).

ENTERING A D V A N C E

Calling bearing_analizer.
Calling target_posit.

ENTERING    A W A I T


ENTERING A D V A N C E

Entering IDPOSIT LOOP.

ENTERING    A W A I T


ENTERING A D V A N C E


ENTERING    A W A I T


ENTERING A D V A N C E

LEAVING CORRELATION
ENTERING CORRELATION

ENTERING    A W A I T

Extracting data from a buffer shared with CORRELA   TE.

ENTERING A D V A N C E


ENTERING    A W A I T

BUFFER FILLED. ADVANCING DISPLAY.

ENTERING A D V A N C E


ENTERING    A W A I T

ENTERING DISPLAY LOOP.
TRACK_NR  X_POSIT   Y_POSIT   TIME      COURSE    SPEED
00        231.8     678.9     0067      240       0345
01        443.1     444.4     0107      240       0345
02        443.1     444.4     0006      240       0345

ENTERING A D V A N C E


ENTERING    A W A I T


Entering IDLE PROCESS.
Entering IDLE PROCESS.
CALL ADVANCE(ID$POSIT).
```

```
ENTERING A D V A N C E

Calling bearing_analizer.
Calling target_posit.

ENTERING    A W A I T


ENTERING A D V A N C E

Entering IDPOSIT LOOP.

ENTERING    A W A I T


ENTERING A D V A N C E


ENTERING    A W A I T


ENTERING A D V A N C E

LEAVING CORRELATION
ENTERING CORRELATION

ENTERING    A W A I T

Extracting data from a buffer shared with CORRELA   TE.

ENTERING A D V A N C E


ENTERING    A W A I T

BUFFER FILLED. ADVANCING DISPLAY.

ENTERING A D V A N C E


ENTERING    A W A I T

ENTERING DISPLAY LOOP.
TRACK_NR  X_POSIT   Y_POSIT   TIME      COURSE    SPEED
00        231.8     678.9     0067      240       0345
01        443.1     444.4     0107      240       0345
02        443.1     444.4     0006      240       0345
03        333.9     335.2     0175      240       0345

ENTERING A D V A N C E


ENTERING    A W A I T

Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
CALL ADVANCE(ID$POSIT).

ENTERING A D V A N C E

Calling bearing analizer.
```

```
Calling target_posit.
ENTERING    A W A I T

ENTERING A D V A N C E

Entering IDPOSIT LOOP.
ENTERING    A W A I T

ENTERING A D V A N C E

ENTERING    A W A I T

ENTERING A D V A N C E

LEAVING CORRELATION
ENTERING CORRELATION
ENTERING    A W A I T

Extracting data from a buffer shared with CORRELA   TE.
ENTERING A D V A N C E

ENTERING    A W A I T

BUFFER FILLED. ADVANCING DISPLAY.
ENTERING A D V A N C E

ENTERING    A W A I T

ENTERING DISPLAY LOOP.
TRACK_NR  X_POSIT   Y_POSIT   TIME      COURSE    SPEED
00        231.8     678.9     0067      240       0345
01        443.1     444.4     0107      240       0345
02        443.1     444.4     0006      240       0345
03        333.9     335.2     0175      240       0345
04        443.1     444.4     0242      240       0345
ENTERING A D V A N C E

ENTERING    A W A I T

Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
Entering IDLE PROCESS.
```

EXAMPLE #6

This is the last and more powerfull example. Last because no time is left for more testing of the operating system. Powerfull because PROCO1 is loaded to one physical processor and PROCO2 is another physical processor. These two interactive processes use the synchronization and communication mechanism and also the hardware interrupt structure as it was configured to provide inter-real processor communication (to support the preemptive scheduling). Furthermore in this example only the ITC level is used to demonstrate the verifiable "loop free" structure of the operating system (the TC level is not linked with the operating ystem in this specific example).

These two processes are initialized as ITC processes, in the same way as they initialized the MMGT and IDLE process. For their interactions, the inter-virtual processor synchronization mechanism is used.

The address space descriptor (the base of the stack) for the IDLE process is 5000, for the MMGT process is 5500, for the first process is 6000 and finally for the second process is 7000. Four VP's are used per real processor.

The input source code for these two processes is under the header EXAMPLE #6 INPUT.

Figure 57 illustrates some more output messages incorporated in new modules used for this current demonstration of the operating system (which were not included in Figure 55).

The output to the printer is included under the header EXAMPLE #6 OUTPUT #1 for the first mocrocomputer and EXAMPLE #6 OUTPUT#2 for the second one.

It can be seen from OUTPUT #2 that when the PROC#2 is waiting the occurrence of the event FLDES (to reach a specified value) because no other process is located on the physical processor, it starts executing the IDLE process (it outputs repeatedly the message ENTERING UPDATECOUNTER). When this specific event reaches the necessary value PROC#1 signals to PROC#2 (using the ADVANCE operation) the occurrence of this event. Since PROC#2 is loaded on another physical processor, the hardware interrupt structure is used to awaken this process (in OUTPUT #1 after the message ITC$ADVANCE there is the message ENTERING HARDWARE$INT). After receiving this signal, the physical processor exits the IDLE process and continues on the previous task (PROC#2) and so on.

To ensure in this example that the hardware interrupt mechanism will be invoked, a different delay is used in these two processes.

LEVEL	MODULE	OUTPUT MESSAGE
	ITC$INIT	ENTERING ITC$INIT
	KERNEL$INIT	ENTERING KERNEL$INIT
	ITC$AWAIT	ENTERING ITC$AWAIT
	ITC$LOCATE$EVC	ENTERING ITC$LOCATE$EVC
	ITC$ADVANCE	ENTERING ITC$ADVANCE
	GET$COUNTER	ENTERING GETCOUNTER
	UPDATE$COUNTER	ENTERING UPDATECOUNTER
	HARDWARE$INT	ENTERING HARDWARE$INT

FIGURE 57. MORE OUTPUT MESSAGES

EXAMPLE #6 INPUT

```
      /*       FILE    PROC21.SRC      18 MAY   */

CSUPPSMODULE: DO;

     DECLARE   (I,Z) BYTE;
     DECLARE    CR  LITERALLY '0DH',
                LF  LITERALLY '0AH';

     DECLARE S WORD;

     DECLARE CSUPP BYTE DATA(33);
     DECLARE FLDES BYTE DATA(44);

DECLARE
 MSG1(*) BYTE INITIAL ('PROC#1. INITIAL ENTRY INTO CLUTTER SUPPRESSION'),
 MSG2(*) BYTE INITIAL ('PROC#1. WAIT FOR DATA READY                   '),
 MSG3(*) BYTE INITIAL ('PROC#1. PERFORMING CLUTTER SUPPRESSION: FRAME #'),
 MSG4(*) BYTE INITIAL ('PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT      ');

 ITC$AWAIT1: PROCEDURE(EVC$ID,AWAITED$VALUE) EXTERNAL;
       DECLARE EVC$ID  BYTE,
               AWAITED$VALUE WORD;
 END;

 ITC$ADVANCE: PROCEDURE(EVC$ID) EXTERNAL;
       DECLARE EVC$ID BYTE;
 END;

     OUT$CHAR: PROCEDURE(CHAR);
        DECLARE CHAR BYTE;
        DO WHILE (INPUT(0DAH) AND 01H) = 0; END;
        OUTPUT(0D8H) = CHAR;
     END;

     OUT$HEX: PROCEDURE(B);
        DECLARE B BYTE;
        DECLARE ASCII(*) BYTE DATA ('0123456789ABCDEF');
        CALL OUT$CHAR(ASCII(SHR(B,4) AND 0FH));
        CALL OUT$CHAR(ASCII(B AND 0FH));
     END;

     I = 2;

        DO Z = 0 TO 45;
          CALL OUT$CHAR(MSG1(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);

     DO WHILE (I <= 0FFH);

        DO Z = 0 TO 45;
          CALL OUT$CHAR(MSG2(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);

        CALL ITC$AWAIT1(CSUPP,I);

        I = I + 1;    /*  <========  */
```

```
            DO Z = 4 TO 47;
            CALL OUTSCHAR(MSG3(Z));
            END;
            CALL OUTSHEX(I);
            CALL OUTSCHAR(CR);
            CALL OUTSCHAR(LF);

            DO K = 0 TO 1000;
              CALL TIME(250);
            END;

            DO Z = 0 TO 45;
               CALL OUTSCHAR(MSG4(Z));
            END;
            CALL OUTSCHAR(CR);
            CALL OUTSCHAR(LF);

            CALL ITC$ADVANCE(FLDES);

         END; /* WHILE */

      END; /* MODULE */
```

```
      /*       FILE    PROC02.SRC      16 MAY   */

FLDES$MODULE: DO;

     DECLARE    I BYTE;
     DECLARE    CR  LITERALLY '0DH',
                LF  LITERALLY '0AH';

     DECLARE Z BYTE;

     DECLARE CSUPP BYTE DATA(33);
     DECLARE FLDES BYTE DATA(44);

DECLARE
 MSG1(*) BYTE INITIAL ('PROC#2. INITIAL ENTRY INTO FILTER DESIGN      '),
 MSG2(*) BYTE INITIAL ('PROC#2. WAIT FOR DATA READY                   '),
 MSG3(*) BYTE INITIAL ('PROC#2. PERFORMING FILTER DESIGN ON FRAME # '),
 MSG4(*) BYTE INITIAL ('PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT');

 ITCSAWAIT1: PROCEDURE(EVC$ID,AWAITED$VALUE) EXTERNAL;
       DECLARE EVC$ID BYTE,
                AWAITED$VALUE WORD;
     END;

 ITCSADVANCE: PROCEDURE(EVC$ID) EXTERNAL;
       DECLARE EVC$ID BYTE;
     END;

     OUT$CHAR: PROCEDURE(CHAR);
        DECLARE CHAR BYTE;
        DO WHILE (INPUT(0DAH) AND 01H) = 0; END;
        OUTPUT(0D8H) = CHAR;
     END;

     OUT$HEX: PROCEDURE(B);
        DECLARE B BYTE;
        DECLARE ASCII(*) BYTE DATA ('0123456789ABCDEF');
        CALL OUT$CHAR(ASCII(SHR(B,4) AND 0FH));
        CALL OUT$CHAR(ASCII(B AND 0FH));
     END;

     I = 0;

        DO Z = 0 TO 45;
          CALL OUT$CHAR(MSG1(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);

        DO WHILE (I <= 0FFH);

        DO Z = 0 TO 45;
          CALL OUT$CHAR(MSG2(Z));
        END;
        CALL OUT$CHAR(CR);
        CALL OUT$CHAR(LF);

        CALL ITCSAWAIT1(FLDES,I);

        I = I + 1;
```

```
            DO Z = 2 TO 43;
              CALL OUT$CHAR(MSG3(Z));
            END;
            CALL OUT$HEX(I);
            CALL OUT$CHAR(CR);
            CALL OUT$CHAR(LF);

            DO Z = 2 TO 100;
               CALL TIME(250);
            END;

            DO Z = 0 TO 45;
              CALL OUT$CHAR(MSG4(Z));
            END;
            CALL OUT$CHAR(CR);
            CALL OUT$CHAR(LF);

            CALL ITC$ADVANCE(CSUPP);

         END; /* WHILE */

      END; /*MODULE */
```

EXAMPLE #6 OUTPUT .1

```
ENTERING ITC$INIT
ENTERING KERNEL$INIT
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  00
            SELECTED$DBR =  0550
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  00

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  00
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
PROC#1. INITIAL ENTRY INTO CLUTTER SUPPRESSION
PROC#1. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. PERFORMING CLUTTER SUPPRESSION: FRAME # 01
PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC

ENTERING  H A R D W A R E $ I N T
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. PERFORMING CLUTTER SUPPRESSION: FRAME # 02
PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING  I T C $ A D V A N C E
```

```
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC

ENTERING  H A R D W A R E $ I N T
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. PERFORMING CLUTTER SUPPRESSION: FRAME # 03
PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC

ENTERING  H A R D W A R E $ I N T
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. PERFORMING CLUTTER SUPPRESSION: FRAME # 04
PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC

ENTERING  H A R D W A R E $ I N T
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  01
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  01
            SELECTED$DBR =  0600
PROC#1. PERFORMING CLUTTER SUPPRESSION: FRAME # 05
PROC#1. ADVANCE FILTER DESIGN EVENTCOUNT
```

EXAMPLE #6 OUTPUT #2

```
ENTERING ITC$INIT
ENTERING KERNEL$INIT
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  04
            SELECTED$DBR =  0550
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  04

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  04
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
            SELECTED$DBR =  0700
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
PROC#2. INITIAL ENTRY INTO FILTER DESIGN
PROC#2. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
            SELECTED$DBR =  0700
PROC#2. PERFORMING FILTER DESIGN ON FRAME # 01
PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
            SELECTED$DBR =  0700
PROC#2. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
            SELECTED$DBR =  0500
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07
ENTERING GETCOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
```

```
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER  ENTERING LOCKVPM
ENTERING RDYTHISVP
ENTERING ITC$RET$VP
         RUNNING$VP$ID =  07
   SET VP TO READY:  VP =  07
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
          SELECTED$DBR =  0700
PROC#2. PERFORMING FILTER DESIGN ON FRAME # 02
PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
         RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
          SELECTED$DBR =  0700
PROC#2. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
         RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
          SELECTED$DBR =  0500
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
         RUNNING$VP$ID =  07

ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING LOCKVPM
ENTERING RDYTHISVP
ENTERING ITC$RET$VP
         RUNNING$VP$ID =  07
   SET VP TO READY:  VP =  07
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
          SELECTED$DBR =  0700
PROC#2. PERFORMING FILTER DESIGN ON FRAME # 03
PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
         RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
          SELECTED$DBR =  0700
PROC#2. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
         RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
```

```
ENTERING GETWORK
          SELECTED$DBR =  0500
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDAENTERING LOCKVPM
ENTERING RDYTHISVP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07
   SET VP TO READY:  VP =  07
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
          SELECTED$DBR =  0700
PROC#2. PERFORMING FILTER DESIGN ON FRAME # 04
PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
          SELECTED$DBR =  0700
PROC#2. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
          SELECTED$DBR =  0500
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07
T
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING LOCKVPM
ENTERING RDYTHISVP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07
   SET VP TO READY:  VP =  07
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
          SELECTED$DBR =  0700
PROC#2. PERFORMING FILTER DESIGN ON FRAME # 05
PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING ITC$ADVANCE
```

```
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
           SELECTED$DBR =  0700
PROC#2. WAIT FOR DATA READY

ENTERING  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
           SELECTED$DBR =  0500
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTERENTERING LOCKVPM
ENTERING RDYTHISVP
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07
   SET VP TO READY:  VP =  07
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
           SELECTED$DBR =  0700
PROC#2. PERFORMING FILTER DESIGN ON FRAME # 06
PROC#2. ADVANCE CLUTTER SUPPRESSION EVENTCOUNT

ENTERING  I T C $ A D V A N C E

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
 SET VP TO RUNNING:  VP =  05
           SELECTED$DBR =  0700
PROC#2. WAIT FOR DATA READY

ENTERIN  I T C $ A W A I T

ENTERING ITC$RET$VP
          RUNNING$VP$ID =  05
ENTERING ITC$LOCATE$EVC
ENTERING GETWORK
           SELECTED$DBR =  0500
ENTERING UNLOCKVPM
ENTERING CHECKPREEMPT
ENTERING ITC$RET$VP
          RUNNING$VP$ID =  07

ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
```

```
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
ENTERING UPDATECOUNTER
```

.
.
.

LIST OF REFERENCES

1. Intel Corporation, PL/M-86 Programming Manual, 1978.

2. Intel Corporation, iSBC 86/12A Single Board Computer Hardware Reference, Manual 1979.

3. Intel Corporation, MCS-86 Software Development Utilities Operating Instructions for ISIS-11 Users, 1979.

4. Intel Corporation, MSC-86 Macro Assembly Language Reference Manual, 1979.

5. Intel Corporation, MSC-86 Macro Assembler Operating Instructions for ISIS-11 Users, 1979.

6. Intel Corporation, ISIS-11 Pl/M-86 Compiler Operator's Manual, 1979.

7. O'Connell, J. S. and Richardson, L. D., Distributed, Secure Design for a Multi-Microprocessor Operating System, Naval Postgraduate School, 1979.

8. Wasson, W. J., Detailed Design of the Kernel of Real-Time Multiprocessor Operating System, Naval Postgraduate School, 1980.

9. Organick, E. I., The Multics System: An Examination of its Structure, The MIT Press, Cambridge, Massachusetts, 1972.

10. Reed, D. P. and Kanodia, R. J., "Synchronization with Eventcounts and Sequencers", Communication of the ACM, v. 22, p. 115-123, February 1979.

11. Dijkstra, E. W., "Cooperating Sequential Processes", in Programming Languages, F. Guneys, ed., Academic Press, 1968.

12. Dijkstra, E. W., "The Structure of the 'THE' Multi-programming System", Communications of the ACM, v. 11, p. 341-346, May 1968.

13. Madnick, S. F. and Donovan, J. J., Operating Systems, McGraw Hill, 1974.

14. Saltzer, J. H., Traffic Control in a Multiplexed Computer System, Ph.D. Thesis, Massachusetts Institute of Technology, 1966.

15. Reed, D. P., Processor Multiplexing in a Layered Operating System, M.S. Thesis, Massachusetts, Institute of Technology, MIT/LCS/TR-164, 1976.

16. Daley, R. G. and Dennis, J. B., "Virtual Memory, Processes, and Sharing in Multics", Communications of the ACM, v. 11, p. 306-312, May 1968.

17. Dijkstra, E. W., "The Humble Programmer", Communications of the ACM, v. 15, No. 10, p. 859-866, October 1972.

18. Anderson, G. A. and Jensen, E. D., "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", Computing Surveys, v. 7, no. 4, p. 197-213, December 1975.

19. Anderson, J. L., Design of a System Initialization Mechanism for a Multiple Microprocessor Computer System, M.S. Thesis, Naval Postgraduate School, 1980.

20. Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", CACM 15, 12, December 1972, pp. 1053-8.

21. Intel Corporation, iSBC 957 INTELLEC-ISBC 86/12 Interface and Execution Package User's Guide, 1978.

22. Schell, LT.Col., R. R., "Security kernels: A Methodical Design of System Security", USE Technical Papers (Spring Conference, 1979). pp. 245-250, March 1979.

23. Anderson, J. P., "Computer Security Technology Planning Study", ESD-TR-73-51, October 1972.

24. Schell, LT.Col. R. R., "Computer Security: the Achilles Heel of the Electronic Air Force?", Air University Review, v. 30, no. 2 p. 16-33, January 1979.

25. Courtois, P. J., Heymans, F., and Parnas, D. L., "Concurrent Control with "Readers" and "Writers", Communications of the ACM, v. 14, no. 5 p. 667-668, October 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 62 Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	1
4.	Department Chainman, Code 52 Department of Comupter Science Naval Postgraduate School Monterey, California 93940	1
5.	Professor T. F. Tao, Code 62Tv Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	7
6.	Associate Professor U. R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93940	3
7.	Associate Professor R. R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
8.	Associate Professor M. L. Cotton, Code 62Cc Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	1
9.	LT Demosthenis K. Rapantzikos, H.N. Karaoli 7, Salamis, Nisos Salamis Greece	5

10. LT Warren J. Wasson, USN 1
Commander Naval Electronics Systems Command
PME 124, National Center 1
Washington, D.C. 20360

11. KUBA (HAIM) 1
86/A Hatishbi Carmel
Haifa, Isreal

12. LT Richard L. Anderson, USN 1
Commander Naval Military Personnel Command
(NMPC - 16F1)
Washington, D.C. 20370

END

DATE
FILMED
10-81

DTIC